UNIVERSITY OF THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND BIOMEDICAL INFORMATICS

Enabling blockchain interaction for IoT devices, through a secure proxy mechanism

Filisia Melissari

Thesis

Advisor

Georgios Spathoulas

Laboratory teaching staff

Lamia, 2021

UNIVERSITY OF THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND BIOMEDICAL INFORMATICS

Enabling blockchain interaction for IoT devices, through a secure proxy mechanism

Filisia Melissari

Thesis

Advisor

Georgios Spathoulas

Laboratory teaching staff

Lamia, 2021

Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. *Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*
2. *Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*
3. *Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια*
4. *Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.*

Ημερομηνία: 16/03/2022

Η Δηλ.

Φιλησία Μελισσάρη

(Υπογραφή)

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.

Enabling blockchain interaction for IoT devices, through a secure proxy mechanism

Filisia Melissari

The committee:

Georgios Spathoulas, Laboratory teaching staff (advisor)

Athanasios Kakarountas, Professor

Petros Spachos, Assistant Professor

ABSTRACT

Title: Creating an embedded system for quick and efficient sending of data in Blockchain applications

Finding a way to send data securely on IoT devices is a challenge that has been given many solutions during the last years. With more embedded applications coming up each year, ranging from agriculture to naval technology, the need for a highly efficient and robust system to do so is needed now more than ever. Blockchain technology being a decentralized, secure, and novel way of sharing information, stands as an interesting answer. This paper proposes the use of a proxy server to help embedded systems interact with smart contracts in a quick and secure way. Using a Pycom module as a device that can be registered and, through the server be able to sign any Ethereum smart contract..

KEYWORDS: blockchain, embedded system, proxy server, security

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

During the last years, the corporate world, as well as the academic, have experienced a new trend–incorporating blockchain with a large variety of applications–and for a good reason. Distributed apps are more secure against cyberattacks, they can be trusted resources since they are not owned by anyone, and their data cannot be lost. Another trend in the realm of computer science, is the Internet of Things; from smart homes to smart cities, automated cars, and novel ways to do agriculture, IoT can be found everywhere. Considering the impact these two trends have on modern society, their combination is not less interesting.

While there have been proposed architectures that combine embedded systems with blockchain, they usually serve very specific purposes and do not truly enable the devices to communicate freely with blockchain. The reason that such an application does not yet exist, is because it is not without challenges. The storage and cost restrictions that come with distributed apps, the different sets of restrictions that may come with embedded systems, the finding of a secure channel of communication between IoT and smart contracts, and the handling of such a system in an honest, responsible, and reliable way.

This paper proposes an architecture that tries to surpass the above hurdles and enable for the first time IoT devices to communicate securely with any smart contract deployed in a network. Specifically, it designs an architecture, analyzes each component, and provides a demo implementation that succeeds in proving the concept.

2. BACKGROUND

When Satoshi Nakamoto first introduced blockchain to the world no one could imagine how much of an impact it could have on society. Today, thirteen years later,

blockchain is everywhere; with nearly 6000 cryptocurrencies known to existence, a whole new voting mechanism, and a huge range of innovative applications used in banking or even healthcare. But what is Blockchain?

2.1 BLOCKCHAIN

Blockchain was first introduced as a mechanism in the Bitcoin white paper. Bitcoin was the first cryptocurrency created and described as a peer-to-peer electronic cash system [1]. The motive was to create a completely distributed banking system that would for the first time remove any third parties behind transactions, and in doing that reduce a lot of the cost, but at the same time remain secure, transparent, and attack-proof.

Blockchain describes a distributed digital ledger that stores data of any kind [2]. Being a distributed system, it is made of a series of nodes, with every node containing the same data, and having the same rights and authorities, so that they are all equal. These nodes act as the servers of the Blockchain network and they control and maintain the ledgers. These can be considered as data stores that contain the same records. The definition of this mechanism is Distributed Ledger Technology [3].

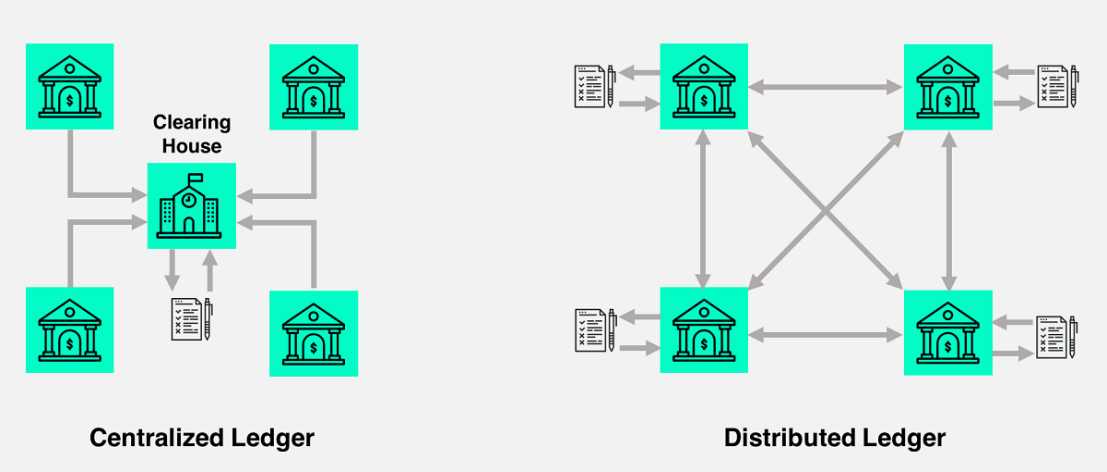

Figure 1 Differences between centralized and distributed ledger [3]

In figure 1, there can be seen the differences between a centralized and distributed ledger. While in the centralized, every communication goes through a central authority, that has more rights and is the only node that holds information about the ledger, in the distributed, things are far different and follow the technology described above. Communication goes in every direction and records are shared. The mechanism of these systems is transparent to every user, as is every record and action within the network. So, it provides a sense of security that no malicious actions will go unnoticed. Furthermore, if a node is under attack and its records compromised, the damage caused will be the least in comparison to the centralized system. The network itself and its data will be intact and functioning normally. If there was a central authority that handled every function, and that authority got attacked, that system would stop working completely and the damage could be irreplicable.

Using blockchain to describe such a system is not accidental, the ledgers form a chain that is made of individual blocks of data, and blockchain is indeed a chain made of blocks.

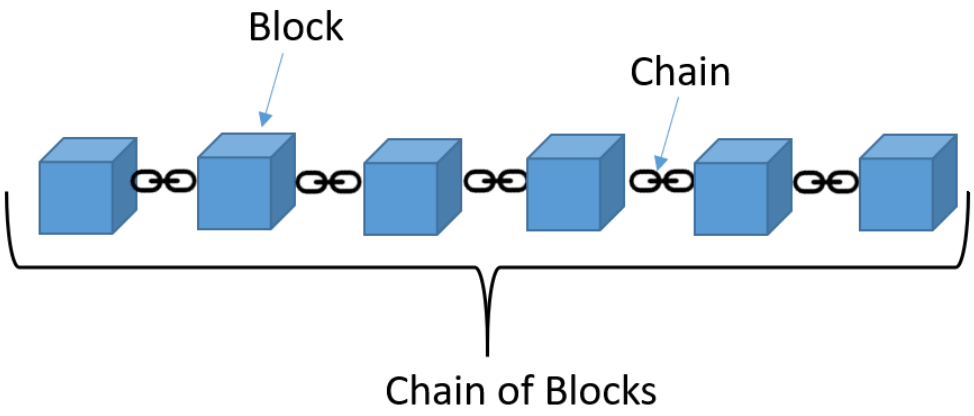

Figure 2 Showcase of the origin of Blockchain as a term [4]

The most common thing stored in those blocks is transactions. In the blockchain realm, a coin is in fact, a chain of transactions. And each owner of coins holds an address, a public, and a private key. For a transaction to be made, the owner of the coin must digitally sign the previous transaction and the public key of the person that the coin is being sold to and finally add these at the end of the coin's chain. Then the person receiving the coin has to verify the transaction by verifying the digital signature with his private key.

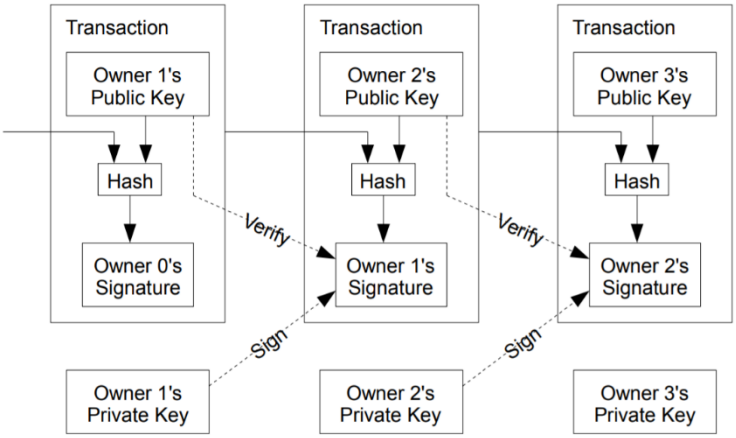

Figure 3 Transaction verification workflow [1]

This model seems to have a problem. What if an owner sells the same coin to two people? Would there be a way for the payees to know that the coin was not double spent? Satoshi Nakamoto said there is. The solution lies in publicly announcing every transaction and every network participant agreeing on the history of the transactions. So, in other words, every party has to have the complete transactions' records at any time.

The way that all the nodes can do that, is through a consensus mechanism. "A consensus mechanism is a fault-tolerant mechanism used in a blockchain to reach an agreement on a single state of the network among distributed nodes. These are protocols that make sure all nodes are synchronized with each other and

agree on transactions, which are legitimate and are added to the blockchain." As described by Aggawarl et al. [5] So, there are protocols that networks apply in order to ensure that information is distributed correctly. They are doing that by verifying the state of the transactions. Depending on the focus between speed, storage, safety, and accuracy. To mention a few, proof-of-work (PoW), proof-of-stake (PoS), delegated proof-of-stake (DPoS), practical Byzantine fault tolerance (PBFT), proof-of-capacity (PoC), proof-of-activity (PoA), proof-of-publication (PoP), proof-of-retrievability (PoR), proof-of-importance (PoI), proof-of-burn (PoB), proof-of-elapsed time (PoET), and proof-of-ownership (PoO). The most commonly used ones are proof-of-work a proof-of-stake.

Proof-of-work is the protocol used by two of the most famous cryptocurrencies in the world; Bitcoin and Ethereum. It describes the way that nodes gain the ability to add new blocks to the chain; they solve complex cryptographic puzzles. The process of solving is called mining, and it involves searching for a value that when hashed is going to start with a specific number of zero bits. The first miner to solve that problem and add a block is given a reward. Based on that system, if a miner wanted to add a malicious block, he had to be the best in solving the puzzle. And to be more precise, he would have to own a computer 51% more powerful than the rest of the network.[6] While that number might seem high enough for now, in the near future, things are not that certain, with the emergence of quantum supremacy. And despite the elevated level of security it provides, it is relatively slow and expensive. These are the reasons that many cryptocurrencies are moving away from the costly and outdated proof-of-work to newer protocols.

Proof-of-stake on the other hand does not involve competition. Instead, each user is assigned a stake, and the creator of the block is chosen randomly by an algorithm based on that stake. The more coins a user owns, the bigger the stake he has. The user that has staked his coins is called a validator and it is analogous to the miner of the PoW mechanism. With this consensus mechanism any barriers

relating to equipment and electricity costs, disappear, so it is easier to become a validator. That means that there should be more nodes and therefore a better decentralized network.

2.2 ETHEREUM

Ethereum was the first blockchain network that came with a complete programming language. It gave users the freedom to write their own smart contracts and create distributed applications.

2.2.1 SMART CONTRACTS

Smart contracts are programs that run on the blockchain. They consist of code and data, and they exist at a specific address on the network. Since they do have an address, they also own a balance that is controlled by the functions of the contract. Regular users can interact with them through their functions and make transactions, but they cannot delete them. From the moment a contract is deployed on the network it can never be deleted. Any person can write a smart contract and deploy it. The only perquisites are the right number of Ethers, and knowledge of Solidity or any other Ethereum smart contract programming language. In fact, a deployment is basically a transaction. To write a smart contract to the blockchain, the gas cost is estimated based on the code written and the storage that it is going to use. After it is calculated, the owner of the contract makes an ether transaction based on the cost and the deployment completes. Ethereum gave wings to smart contract development and created a new era for distributed apps, but in reality, smart contracts are just a type of account.

2.2.2 ACCOUNTS

Accounts are entities that exist on the Ethereum network, they own balance, and they can send transactions. There are two types of accounts; externally owned which are controlled by users, and smart contracts. Both types can send and receive ethers and interact with smart contracts. Their differences lie in certain aspects of their functionality; while creating an account is free, smart contract creation, as explained in the above paragraph, does cost. Furthermore, contracts cannot initiate transactions themselves, they can only send transactions as a response triggered by their code. On the other hand, users' accounts cannot perform such transactions, only ether or token transfers.

On a more technical aspect, an account is a 20-byte address and is represented by four fields: the nonce, the balance, a storage hash, and a code hash. The nonce is the holder of information about how many transactions have been made from the account. Its existence is important since it prevents processing the same transaction twice. The balance is the number of wei that belong to the address. Before dwelling on to explain the next field, storage hash, it is important to explain a certain type of component that is essential; the Merkle tree. This tree is a hash tree, where every leaf is a hash of a data block, and every node is a hash of the labels of its child nodes' hashes. It is used to perform verifications on the contents of large data structures securely and efficiently [8]. Say we make a transaction and let it be called T_A and H_A the equivalent hash. The H_A hash of the transaction will be stored on a leaf in the merkle tree. And consider another transaction T_B, and the hash H_B stored on another leaf. Together, the adjacent transactions are getting hashed and form a new hash, their parent node, which represents both transactions. The process can be repeated until the last hash value is created and it is called the merkle root. This root holds information about every transaction and provides easy access for their verification, so the process becomes simple, and the nodes do not need to download a huge amount of data, but only block headers. So, the storage hash is the 256-bit hash of the root node of the

merkle tree of a single account, it encodes its storage contents, and it is empty by default.

And lastly, the code hash, that while it exists as a field for both types of accounts, for externally owned ones is an empty string, and the reason being is that it only makes sense in the use of smart contracts. This hash refers to the code of the account on the Ethereum Virtual Machine that gets executed every time the contract gets a message call. In contrast to other fields, it is unchangeable. [7]

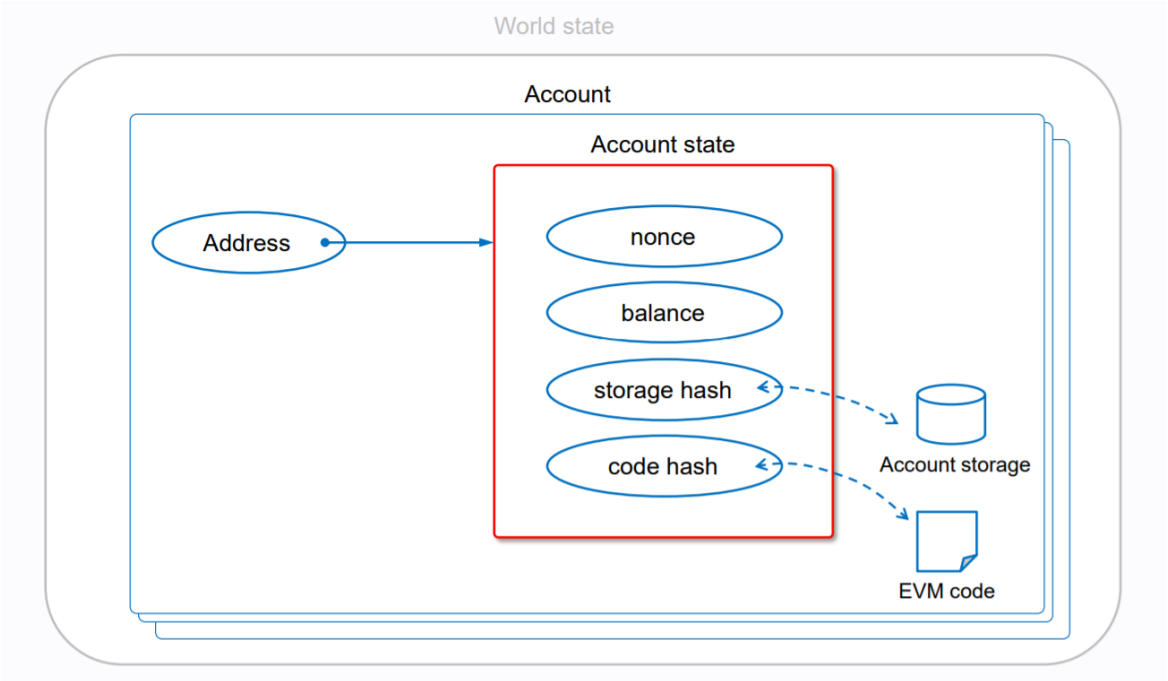

Figure 4 Components of an account [9]

Another component that belongs to an account is a pair of cryptographic keys; a public key used by others to make a transaction with a certain address, and the private key that is used to sign the transaction and gives the user ownership of the funds he holds in his account.

2.2.3 TRANSACTIONS

The Ethereum transactions in reality are signed data packages that contain messages. The data include information about the signature of the sender, the address of the recipient, the amount of ether to be transferred, some data if they exist, and two specific values: STARTGAS and GASPRICE. While the prior three fields are a standard for every cryptocurrency, the last two were first introduced on the Ethereum network. Each transaction, depending on its parameters and possibly the script behind a smart contract method, takes a certain number of computational steps to be completed. Startgas, refers to the maximum number of such steps, that a transaction is allowed to take. If that number is surpassed, the transaction gets reverted. Also, every such step is computationally costly as it may strain the resources of every node on the network, so a fee is needed to keep the strain in check. That is why the gasprice field was added. It is the value that represents a fee per computational step. The fees system apart from keeping the system light has another reason to exist; it protects from denial-of-service attacks. While smart contract programming was an innovation that made Ethereum popular in the first place, it also expanded the attack window that could be caused by malicious code or even by accidental infinite loops.

2.2.4 MESSAGES

As mentioned before, smart contracts do not have the ability to perform transactions. But another type of similar functionality would be needed for scaling distributed ethereum apps. That is why messages were introduced. Essentially messages are not real objects that are serialized, but they exist only within the execution environment, they are transactions that but with the actors being contracts. They give the ability to contracts to interact with other contracts. They contain the same fields like a normal transaction except for the gasprice.

2.3 EMBEDDED SYSTEMS

Embedded systems are computing systems used for specific purposes that are "embedded" in application environments or other computing systems [10]. They are made of components that usually involve hardware and software. They can be applied in a large variety of fields, like consumer electronics, automotive, military, medical, internet of things, telecommunications, and smart cities.

2.3.1 CHARACTERISTICS

These systems are single-functioned. Meaning that they serve a specific purpose and they do it repeatedly. They cannot be used unchanged for any other operation. Furthermore, they are tightly constrained as per their design metrics. They have to have a specific cost and size, to be fast enough, perform to a certain standard and be able to last if on battery. Also, they are reactive in real-time. They have to react to changes in the system like processing received requests and computing and serving the results in real-time. Moreover, they are microprocessor or microcontroller-based and they must have some type of memory like a computer's ROM, and they usually involve a user or system's interface. As would be expected, in order to work in a system every component has to be connected in some way with at least one other.

Such systems can be easily customized, they usually cost little and consume little power as they might have to work autonomously, and they enhance the performance. On the other side, they are more difficult to develop and market [11].

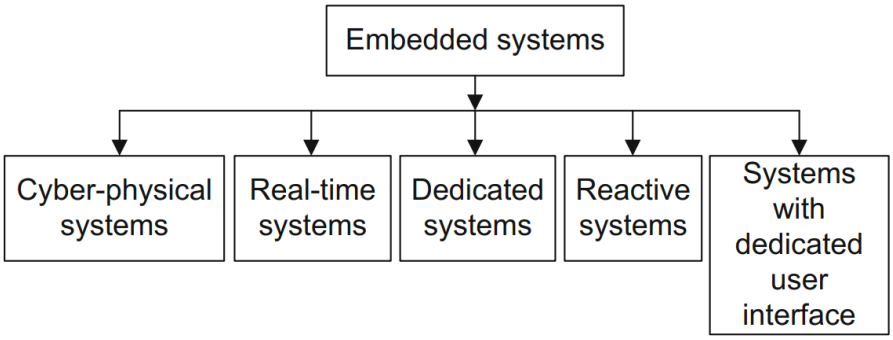

Figure 5 Dependability aspects of embedded systems

2.3.2 REQUIREMENTS

The main requirements of embedded systems are to be dependable and efficient. Being cyber-physical systems, they do interact with the environment and so dependability is a main issue that could be estimated by some key factors: reliability, maintainability, availability, safety, security, and survivability. Dependability reflects the user's degree of trust in the system, and if it is not fulfilled the system could be deemed insecure and unreliable. Dependability could be achieved through having redundant components in case of failure and having secured more than one way to achieve functionality. Regarding the efficiency of the system, it is reached through the equilibrium of performance, power, and cost, and it is measurable in contrast with dependability. The main metrics of valuing efficiency are energy, the optimization for the code size, the run-time efficiency of the resources, the physical weight, and the cost. All of these requirements and subfactors should be taken into account when designing an embedded system.

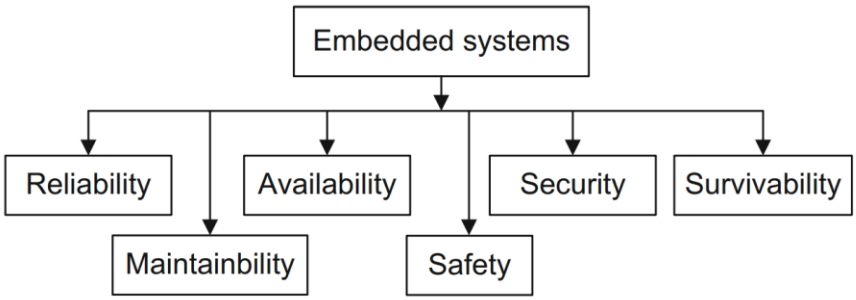

Figure 6 Efficiency aspects of embedded systems

3. RELATED WORK

M.Shurman et al. [12] propose an architectural guideline to enable the usage of blockchain in IoT devices. In that system, IoT devices have been configured to be able to deploy Ethereum smart contracts and rent their service. A client can rent the device by using either a serial number or a QR code that belongs to that device and by doing so they buy the service and take its ownership. The role of smart contracts is to accept offers, manage ownership, and validate the data sent. In this system, someone can only become a client by renting the device, so users cannot use their own devices, and the management of the system is done automatically by a deployed smart contract.

Huh et al. [13], have developed a blockchain platform to manage IoT devices. They built a scenario where there are some physical components, namely a smartphone, and three Raspberry Pis, and these send and retrieve information from the cyber components that are smart contracts on the Ethereum network. The three Raspberries are meters to the electric usage of an air conditioner and a lightbulb, and the smartphone is used to configure an energy policy for the said lightbulb and air conditioner. There are three processes that run concurrently: the user sets the

configuration, and it is sent to Ethereum, the devices receive these data periodically, and they update the contract with the electric usage data they tracked. The proof of concept was successful, and the system run as designed but a couple of weaknesses were unearthed; the difficulty to use the design for time-sensitive purposes, due to the amount of time transactions needed to complete, and the size of the Ethereum client that would force the usage of a third-party proxy.

Samaniego et al. [14], explore the issue of the hosting location of Blockchain and IoT systems and setting cloud and fog platforms as the solution. In this paper, the intention of using Blockchain for IoT is the storage and sharing of data and code. Various experiments were performed to evaluate the performance of using a local cluster of fog and cloud computing as hosts. Two Edison Arduino boards were used and a variable number of clients to write to a multichain. The results showcased that the main issue for performance was not the private multichain but the network chip and traffic of the board. Then, they tried to evaluate the impact of cloud latency in a private blockchain network, specifically IBM's BlueMix. The results revealed a network latency that could be explained by the use of cloud services, and some service failures indicated that BlueMix may not be the best solution when using IoT. The analysis proved network latency as the key factor and fog as the better option compared to the cloud.

Dorri et al. [15], suggest a secure blockchain-based smart home framework. The main focus of the paper is on the security and privacy aspects of the implementation. The architecture consists of three tiers; the smart home which is consisted of smart devices centrally controlled by a miner, the cloud used by devices to store and share data, and an overlay network. In the overlay, nodes are grouped into clusters each with a cluster head that maintains a public blockchain that contains two lists with information about access and authorization. The clusters contain smart homes to which smart devices belong, cloud storage, and a service provider. As per the security aspect, the authors proved that the framework is proofed from DDOS and linking attacks due to its design using the miner and various access controls. To

achieve this level of security additional delays can be caused but, their impact does not affect the availability of the devices. Next, simulations were made to compare the overhead of the system comparing it to a system without encryption, hashing, and blockchain. They evaluated the results on three levels: packet and time overhead and energy consumption. While the performance of the proposed system on all three levels was worse it was deemed insignificant compared to the level of security provided.

Pavithran et al. [16], describe some key characteristics that should be taken into consideration when trying to design the architecture of a blockchain and IoT system, and the related challenges, and point out security gaps in such frameworks. The first challenge described is using different devices under the same blockchain, and the solution proposed would be to standardize the implementation, add policies on data and apply general regulations. Next, as per the type of the blockchain, they determined that permissioned networks would work better with IoT devices as their lighter than permissionless and hybrid. Another challenge mentioned is the volume of the data produced by sensors and the time and storage restrictions blockchain would bring. This could be solved by assigning an AI to clear unwanted data. Also, they mention the importance of securely storing private keys in devices. Furthermore, they set the security requirements and define problems like the usage of asymmetric key cryptography and quantum computing, and IoT data reliability. Finally, they discussed the optimal platform and consensus for IoT and ended up with Hyperledger Fabric and Practical Byzantine Fault Tolerance (PBFT). They concluded that an efficient blockchain platform for IoT does not exist yet.

Another literature review was carried out by Kuang et al [17]. They analyze the solutions that have been proposed by academia, and the methodologies of integration. They covered thirty-five solutions from peer-reviewed academic papers that aimed to surpass the challenges of performance and scalability. On issues based on IoT, they suggested encrypting confidential data before posting them on chain, making smart contracts extensible if they are used as access controls, since as a type

of data they are immutable, adding representation of code on blockchain for integrity and using the PBFT consensus when wanting to achieve real-time monitoring. On the blockchain infrastructure, they recommended permissioned blockchains as they work better with IoT and if employing public networks, to use anchoring mechanisms. For consensus mechanisms, they discouraged the use of proof-of-work and favored proof-of-stake and PBFT. They identified design defects like using other devices to enhance computation and risking single point of failure attacks, provided insight on key factors to consider, and recommended alternatives and solutions.

Christidis et al. [18] also explore the issue of using IoT and blockchain in a single system. They explored the concepts of blockchain and things working together in the preliminary stages when not many such implementations existed, and some of the issues mentioned have been improved or even solved, but some remain, and their importance is as high as ever. One such issue is the performance throughput and high latencies in public blockchain networks, especially when the proof-of-work mechanism is employed. Their research concludes by highlighting the potential that such a combination could have, automating workflows in unique and secure ways and cutting costs in the process, and predicted the spread of these applications in several industries.

Ge et al. [19] proposed a distributed framework that exploits blockchain technology to communicate in a secure way with UAVs (Unmanned Aerial Vehicles), protect them from attacks, and let them keep operational autonomy. They created a new, lightweight blockchain architecture that solves the related issues with performance and storage and manages to maintain the security and privacy-related benefits. To achieve this, they implemented a new structure for blocks and transactions, and a new consensus mechanism inspired by delegated proof-of-stake that involved reputation evaluation for untrusted drones. They analyzed the security of the system and proved its reliability and performed experiments that verified it met throughput data requirements while UAVs retained their self-defense mechanism.

4. SYSTEM ARCHITECTURE

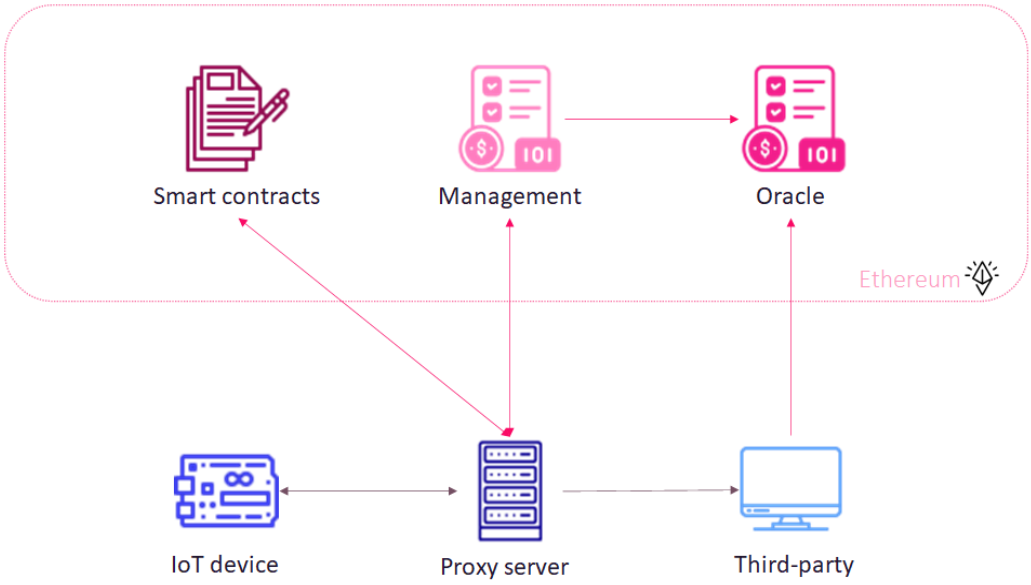

Figure 7 Overview of system architecture

The system architecture involves parties communicating in a decentralized way. These parties exist both in the physical world, like the IoT device and the server, and in the Ethereum realm. The proxy server is the intermediary between the IoT device and the ethereum. There could be several servers all equal to each other, but, for simplicity, it can be considered a single entity. A brief overview of the entire system architecture can be seen in Figure 7. Each component and its role are going to be analyzed in the Components section. The purpose of the system is to function as an enabler for IoT devices to communicate with Blockchain Ethereum applications. A user can register multiple IoT devices and interact with any deployed Ethereum smart contract, and thus the system can have many uses in several sectors, like agriculture and energy retailing.

4.1 COMPONENTS

4.1.1 PROXY SERVER

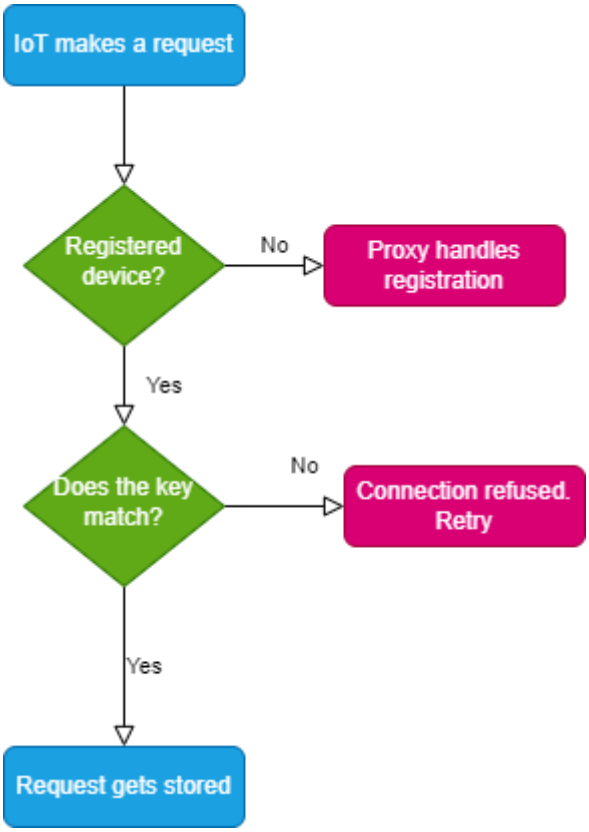

Figure 8 Datagram of an IoT request

The proxy server is the mediator and handler for most of the communications of the system. It interacts with devices, smart contracts, and other third-party applications. As explained above, while for the demonstration there has been developed only one server, in reality, the role and the tasks could have been

distributed between a number of equal servers, and therefore make the system more decentralized.

The server uses stream web sockets and listens for communications from devices. Once communication has been established the server receives the request and checks whether the device trying to connect has been registered or not. In the second case, the server refuses the request and prompts the user to register. In the first case, the proxy starts to process the request and then proceeds to make a raw transaction or a call to a function depending on the request. When the transaction has been completed and the transaction receipt along with any outputs have been retrieved, a message is created that contains the status of the transaction and the outputs if there are any. This message will be sent to the device after a series of verifications. At the same time, another web stream socket service is running, where the server communicates with a third-party application and is sending the transaction hash through the channel. Also, the initial request from the device, along with the transaction hash and the method-id of the function called, are being sent to the management contract that handles transaction costs, and payments. The purpose of this transaction is to request payment for the service provided, namely the gas costs of the interaction and a typical fee. After the successful retrieval of the payment, the proxy can now proceed to send the message created containing the outputs to the device, through the initial socket connection.

4.1.2 DEVICE

The IoT device is the client of the application. The user must first register the device by sharing a public key and then transact a certain amount of ether to the management contract that handles the cost of the transactions. After the registration, the user is free to use the system and interact with his contract of choice by sharing the ABI of the contract with the proxy server and initiating a

request. The request will be containing the id of the device, the public key, the address of the contract, the function which he wishes to use, and the contract parameters if any. The communication with the proxy server is handled using web stream sockets. After the successful receival, process, and completion of the request, the status of the transaction and any outputs will be sent back to the device and can be used accordingly. Also, the owner of the device will be charged with the gas costs and a usage fee that will be covered by the amount deposited into management contract. The rest of the amount can either be withdrawn or remain as is for future usage of the service.

4.1.3 SMART CONTRACT

The smart contract is the contract that the user wants the device to communicate with. It must be deployed to the Ethereum Network and the user must have knowledge of its ABI and address. The size, the functions, and its parameters are independent of the application and thus any contract can be used successfully. For purposes of demonstration, there have been developed and deployed two exemplary contracts.

4.1.4 ORACLE CONTRACT

The oracle contract is providing essential information to the management contract, which is going to be described below, that is needed to certify that the parameters of the transaction made, were indeed those that the user requested, and the transaction was successful. In order to do that, the oracle receives the transaction receipt from the third-party app, like the transaction

hash, the id of the method called, the parameters used, and of course the address of the contract. After successfully storing the information, these can be retrieved by the management contract as needed.

4.1.5 THIRD-PARTY

The role of this application is to provide information about the transaction from a trusted source to the oracle contract. It connects to the proxy server and retrieves the transaction hash. Then, it makes a GET request to the Etherscan page according to the hash, and through a custom API retrieves specific information needed. Having certified that the transaction was completed correctly, it makes a raw transaction to the Oracle contract sending the information. The application is in a way independent of the server. The only information asked is the transaction hash to make the request, and the other party is completely trusted to provide truthful information, making the app itself act as a trusted source for data to the verification process.

4.1.6 MANAGEMENT CONTRACT

The role of this contract is to verify that the transaction was successful, and the data provided were exactly what the device requested, and also, to handle the service payments. The user deposits ethers to this contract to use the system, and it stores information about the user and his current balance. It receives payment requests from the proxy server, processes them, and asks the oracle contract for relevant information about the transaction. Then, it makes a comparison, and if the transaction was successful and the data restored from the Etherscan page that the Oracle holds are the same as those that the device

request contains it can go on to process the payment. To do that, it checks the current balance of the user whose device made the request. If there is enough balance, it withdraws the appropriate amount and informs the proxy server about the completion of the payment. Else, if there is not enough balance, it also informs the proxy accordingly, so that the device cannot receive the results. The contract also provides the selection to the user to withdraw from the amount deposited at any time, making it secure for the users to use the service and not fear monetary losses.

5. IMPLEMENTATION

5.1 TECHNICAL CHARACTERISTICS

For the development of the project the components had the following characteristics:

- Device: Pycom Pysense mounted on an Expansion Board v2.0
- Server: Windows 10/ Intel i7 8th gen/ SSD 256 GB/ 16 GB RAM

5.2 PREREQUISITES

Before diving into the actual operations of the system, some things are assumed to have been completed. Namely, the registration of a user. Normally, the user would have to register first in order to use the application. The registration page would prompt the user to create a username and a password and log into his account to set up needed information. Then the user would have to add unique ids for his devices, the ABIs of all the contracts he would like his devices to interact with, the RSA public keys of his devices that would be used for authentication and add the wallet address that he could use to pay his fees. Next, he would be prompted to deposit to the address of the management contract. With everything having succeeded, registration would have been completed.

The proxy server could manage the registration by saving files posted in directories per user account and saving all the needed user information in a SQL database.

For the current implementation, we have assumed that a user is already registered, all his information exists in a database and his needed files are already in the possession of the proxy server.

5.3 INITIAL REQUEST

Let's consider a user that has a Pycom device in his possession, with the characteristics mentioned above in [4.1], and he intends to use the service to interact with a simple smart contract. To initiate a request, he would have to first build it. The structure of a request is a string that has several fields separated by the character '#'. The fields have a specific order, but their actual number depends on the number of arguments of the function that the user wants to interact with. The basic structure would look like this:

<device_id>#<contract_address>#<function_name>#<argument1>#<argument2>#...#<signature>

The information about the fields can be seen in [Table 1].

Field Name	Description
device_id	The id of the current device that the user added upon registration.
contract_address	The address of the deployed contract that the user wants to interact with.
function_name	The name of the function that the user wants to use.
argument	An argument of the function to be used.
signature	The device_id signed with the private RSA key of the device.

Table 1 Request fields

After the request has been correctly formatted the device has to connect to the proxy server and send the request. To do that, a web socket client has to be initiated and try to connect to the address and port of the server. This can be done by the following code snippet in Micropython using the socket library:

```
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host, port))
```

After successfully connecting, the user has to send the request to the server by:

```
s.sendall(request)
```

Then the request will be received by the proxy server, processed, and eventually completed if every parameter needed is fulfilled. The device will wait to receive the results of the transaction on the established socket connection and then once received close it:

```
reply=s.recv(4096)
s.close()
```

5.4 PROCESS REQUEST

The proxy server is a script written in Python 3.9. The first thing it does is open a port to listen for devices trying to connect and send their requests. A web socket server needs to be built as seen below:

```
serversocket = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM)
serversocket.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR, 1)
```

```
serversocket.bind(("",                                    6544))
while                                                      True:
    (clientsocket, address) = serversocket.accept()
```

In the above snippet, the server sets the appropriate protocols, as AF_INET to accept IPv4 addresses, and SOCK_STREAM to use the stream types of sockets, that are sequenced and two-wayed. Then sets further socket options and binds port 6544 to listen for device requests. Then every time a device tries to connect the server can accept the connection and establish a socket with the device. Then the request is received by using:

```
request = clientsocket.recv(1000)
```

At this stage, the function that processes the request parameters is called, namely decode_requests(). It operates as follows:

1. Splits the request into individual fields
2. Determines if the contract function takes arguments or not and if yes, how many
3. Formats the individual fields from bytes to their appropriate format and especially for the arguments it finds the format by crosschecking with the ABI for their data type
4. Creates the method_id from the function name and arguments.
5. Using the device_id, it retrieves the appropriate information from the database and formats it for use within Web3.
6. Calls the function to make the transaction with all the arguments formatted from the previous steps
7. Creates the reply that will be sent back to the device and contains information about the output of the transaction.

The code of the decode_requests method, as well the complete code of the system can be seen at the end of the paper.

5.5 SEND TRANSACTION

To connect to an Ethereum client, the use of the Web3 library is obliged. It enables users to interact with their ethereum nodes of choice. Clients could be local like Ganache, or remote like in the Ethereum mainnet. For python there is a specific library web3.py, that is going to be used for the application. Also, the contracts have been deployed in an Ethereum testnet called Ropsten. While it is used only for development purposes, it is considered the testnet closest to the main net behavior-wise. Also, to connect to the Ropsten network, a backend and an API provider are needed, for that Infura is recommended. By creating a project in Infura, one can obtain a project key to use to connect to the Ropsten client.

So, the following line of code establishes an ethereum client as discussed:

```
w3_ = Web3(HTTPProvider("https://ropsten.infura.io/v3/cbe831fc4e9a47699e55af9141b7930f"))
```

The URL seen is provided by Infura and it was used in the specific project. Then, a contract instance has to be created, that will be used to interact with it. The contract has to be deployed on the same network using the same Infura key. For the purposes of demonstration, the smart contract that a user wants to use has been created in Solidity with compiler version 0.8.0. The code of the exemplary smart contract can be seen at the end of the paper. So, the creation of the contract instance is in the below snippet:

```
entity = w3_.eth.contract(address=contract_address, abi=abi)
```

Using `entity`, the transaction can be built.

```
func = getattr(entity.functions, func_name)
```

```
nonce = w3_.eth.get_transaction_count(account_address)
```

```
entity_txn = func(*args).buildTransaction({
    'from': account_address,
    'chainId': 3,
    'value': value,
    'gas': 1000000,
    'gasPrice': w3_.toWei('200', 'gwei'),
    'nonce': nonce,
})
```

The first line of code selects the function to be called from the contract instance, based on the name received from the device. On the second line of code, the nonce of the account of the proxy is retrieved, and then the transaction is built on the `entity_txn`. The parameters are described in [Table 2].

Parameter	Description
nonce	The number of completed transactions of an account
chainId	The number of the chain id of the network. For example, for Ethereum mainnet it is 1, for Ropsten Testnet it is 3. For private networks, it can be customized.
value	A number of ether or any denomination to be sent to contract function. The function has to be payable, or else the value has to be zero.
gas	The cost necessary to complete a transaction. It can be estimated from the contract ABI.
gasPrice	The cost of every computational step, or the cost per unit of gas. In proof-of-work in mechanisms, miners prefer to mine transactions with higher gas prices, so the highest this field, the quicker the transaction will be mined.

Table 2 Description of transaction parameters

Afterwards, the transaction is signed using the proxy server's private key and sent using the following lines:

```
signed_txn = w3_.eth.account.sign_transaction(entity_txn,
private_key=priv_key)

tx_hash_b = w3_.eth.send_raw_transaction(signed_txn.rawTransaction)
```

If the type of the function the device defined was view, meaning it did not alter the state of the contract storage, instead of building a transaction to retrieve the output one can just make a function call like this:

```
output = func(*args).call()
```

Where the output will be what the contract function returned.

In the first case, the hash of the transaction is retrieved and the status of the transaction along with other information can be tracked on the Etherscan website for the Ropsten Testnet network [https://ropsten.etherscan.io/]. Also, due to the fact that, as mentioned before, Ropsten acts like the mainnet and the consensus is proof-of-work, there are related latencies that cause a specific problem. Since all transactions are executed from the proxy server's account, they have the same nonce. This means that if synchronization is not handled correctly, they might get reverted due to using the same nonce which will be interpreted as trying to replace a transaction. This problem is handled by waiting for transactions to be completed before the program moves on. To do that, a simple comparison is made to check whether the nonce has changed or not.

5.6 THIRD-PARTY

After the transaction hash has been obtained, the proxy has to notify the third-party application. This application is another script developed in Python 3.9. To get the hash from the server it opens a socket connection on a port in the same way as described in section [4.4]. Then it makes a get request to the Etherscan website, querying with the hash to get the page with information about the transaction. While Etherscan does provide an API to make calls, this did not cover the needs of this application and so a custom API was made. An example of a page as the one requested can be seen in Figure 9. The fields that need to be gathered are: From, To, Value, Transaction Fee, Gas Limit, and Gas Used By Transaction. Their description can be seen in Table 3.

Field	Description
From	The address of the account sending the transaction.
To	The address of the recipient of the transaction, it could be either a contract address or the address of another user.
Value	The amount of ether sent along with the transaction.
Transaction Fee	The fee paid to the miner for processing the transaction.
Gas Limit	Maximum amount of gas a transaction is allowed to consume.
Gas Used by Transaction	The amount of gas the transaction ended up consuming.

Table 3 Description of Etherscan fields used by the Third-Party

Having gathered all the needed fields from Etherscan, the third-party proceeds to send this information to the Blockchain, making a transaction as described in section [4.5]. Specifically, there is a smart contract with the purpose of acting as an oracle and providing necessary information obtained from Etherscan to the Ethereum network. Blockchain networks in general, while allowing for the development of distributed apps, they enforce a lot of restrictions. Due to the nature of the network, data that need to be sourced from somewhere outside of the scope of smart contracts, are unreachable. For example, not even the current time can be retrieved inside the code of a smart contract. To solve this problem, oracles are employed. They act as portals to the real web and serve information needed to the blockchain. For the scope of this application, the third-party app together with the Oracle smart contract feed

information from the trusted source, that is Etherscan to the ethereum network. This data will be needed as a cross-reference to check whether the proxy served the device as intended. It adds a layer of integrity to the system.

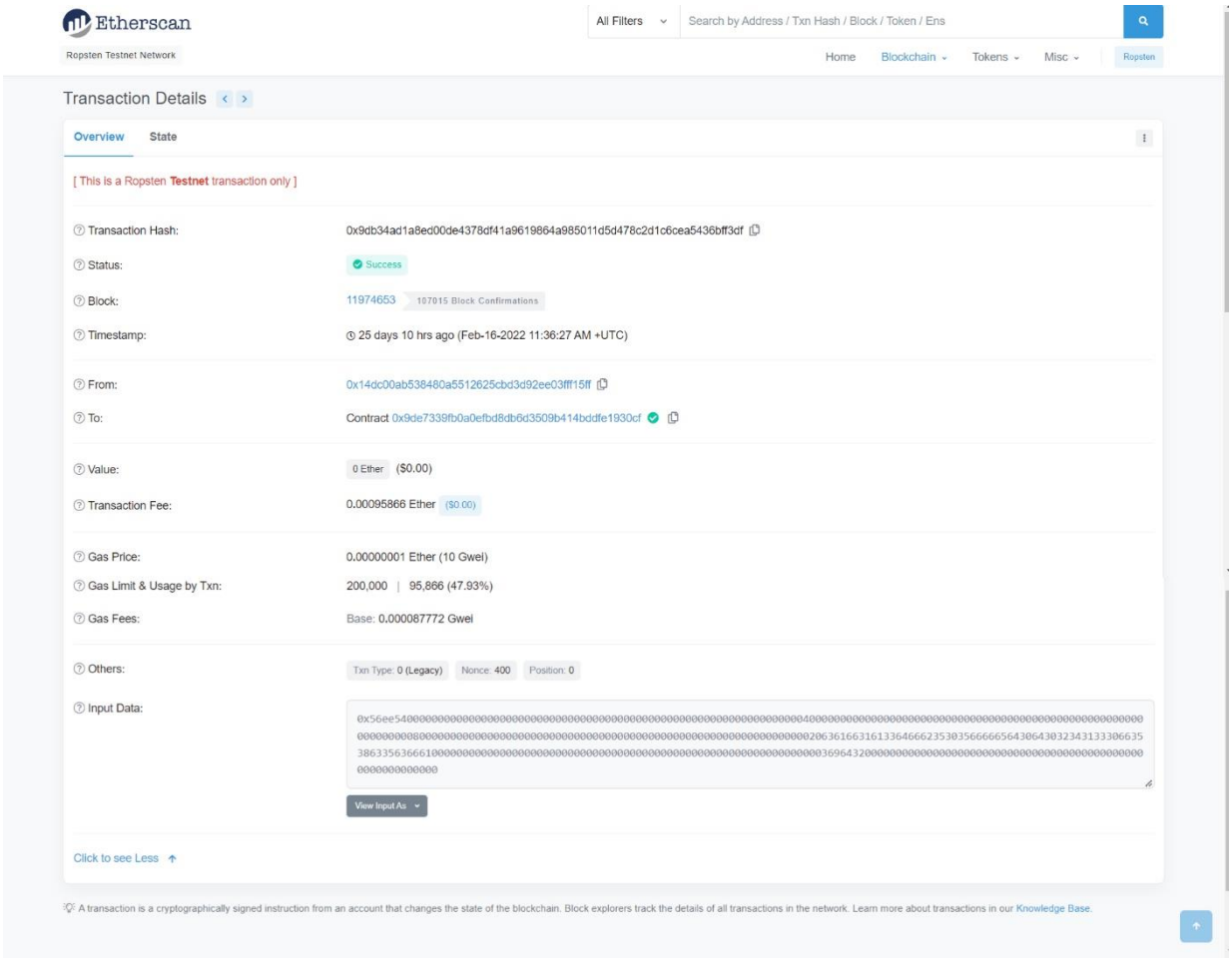

Figure 9 Example of a transaction page from Etherscan

5.6.1 ORACLE

The oracle contract was developed in Solidity version 0.8.0. Its storage consists of a mapping of structs. In solidity mappings are like dictionaries, they are arrays of keys and values. This mapping is pubic, meaning it can be accessed by anyone, and the type of struct it stores is called Receipt and contains information about a transaction that the third-party sent as discussed above. The receipts can be accessed by the transaction hashes. Two simple functions have been implemented;

one to set a receipt, which is the function that the third-party calls to send the data, and one to get a receipt based on a transaction hash, that is used by a contract that will be described later. The full code of the oracle contract is attached at the end.

5.7 REQUEST PAYMENT

The proxy server having sent the transaction hash to the third-party app for verification, proceeds to make a request to be paid for the service provided. To do that it makes a call to a specific purpose contract called Management. The role of this is to make verifications and fulfill requests for payment. It is the contract that the users make deposits to when registering. It also allows for the users to check their balance on the contract and withdraw from their balance if need be.

5.7.1 MANAGEMENT

The contract's storage consists of two mappings, one called device_ids that holds information about to whom does a device belong and deposited_amounts where the current balance of a user on the contract is stored. Also, there are two types of structs, the Request and the Signature as seen below:

```
struct Request{
    string device_id;
    string contract_address;
    string funcname;
    string method_id;
}

struct Signature{
    bytes32 message;
    bytes s;
    bytes e;
    bytes m;
    address payable ver;
}
```

These structs are used to save data sent from the proxy server when making a call to the request_payment function. The first one is used to save the data that the proxy server claims to have used to fill the user's request, along with the device's id. The second one is used for verification; its fields are also sent by the proxy server when making a request and its purpose is going to be analyzed further below. Structs in solidity can be a necessity since they provide a sophisticated solution to a rather big problem. Solidity applies a restriction to the number of variables a user can use in a function. If the threshold is surpassed a compilation error will emerge saying that the stack is too deep. By using structs, the number of variables used is reduced while the volume of data stays the same.

The definition of the request payment function is the following:

```
  function request_payment(string memory request, string memory txhash,
string memory request_method_id, uint numofargs) external payable
returns(uint)
```

The function is external meaning it can read call data directly and doesn't need to copy them to memory first, which means it is not as expensive as defining it as public, and since it is not called internally in the contract that definition is preferred. Also, it is payable. This modifier claims that the function is able to receive ether. The description for the rest of the parameters can be seen in [Table 4].

Parameter	Description
request	The full request as sent from the device
txhash	The hash of the transaction made on behalf of the device
request_method_id	The method_id of the function called on behalf of the device. It is calculated using the sha3 hash of the function definition. e.g., func_name(param1, param2)
numofargs	It is the number of arguments used in the function called and is needed only for processing purposes

Table 4 Description of request_payment function parameters

Solidity does not have implemented string support, so in order to process the fields of the device request, another function has been implemented that splits the string based on a delimiter, which in this case is the character '#'. After processing the fields, it stores them into the structs to be accessed when needed. In order to verify that the request has been fulfilled according to the user's wishes, it needs to crosscheck that the fields of the initial request match the fields used in the actual transaction. So, request_payment function has to make a call to the oracle contract and specifically to the function that returns a Receipt based on a transaction hash. As mentioned in section [1.2.2], smart contracts are allowed to interact with other smart contracts. There are several ways to achieve that depending on the situation. In this case since the code of the Oracle contract is accessible, and the version of Solidity is the same on both contracts, the oracle can be added as an interface to the management contract. Then using its address to initiate a contract instance inside management, and then the functions of the oracle can be accessed directly. In this instance the function call is the following:

```
(string memory tx_fee, string memory oracle_method_id, string memory
oracle_arguments) = o.get_data(txhash);
```

Then the comparison can be made and if successful the contract proceeds to make the rest of the verifications. Another one is to check whether the user has enough ethers deposited to be able to pay for the transaction and service fees, by checking with the device id in the device_ids mapping to find the user's address and then using that to see the deposited amount in the deposited_amounts mapping. If the user indeed has enough, it proceeds to make the last verification, that the device's identity is correct, and it is not someone else trying to act as the device. To do that, asymmetric encryption is employed, and to be precise, the RSA algorithm. The device owns a set of keys and when making a registration it shares the public key with the proxy server. When making a request the device sends along with the actual device_id, the device_id signed with its private key. So, the proxy can then get the exponent and the modulus from the public key and send them along to the

management contract to make the verification. The contract makes use of a library published by [source] that is deployed on the same network. This library has a function that can verify RSA signatures and return zero if they're valid. The code of the library is developed in solidity version 0.6.0, which is different from the version of the management contract. So, to call the library from inside the contract an interface will not work as it did with the oracle contract. Instead, knowing the function definition of the library, it can be called like this:

```
bytes memory payload =
abi.encodeWithSignature("pkcs1Sha256Verify(bytes32,bytes,bytes,bytes)",
_sha256,_s,_e,_m);
      (bool success, bytes memory data) = _addr.delegatecall(payload);
```

Using the function definition `encodeWithSignature` creates the encoding in the appropriate form so that it can be understood by the Ethereum Virtual Machine. Then it proceeds to make a special type of call that is called delegatecall and is used for external libraries. The result of the verification is stored in the data variable.

Finally, if every requirement is fulfilled, it sends the amount of ethers from the total cost of the service and gas fees to the proxy server account, updates the mapping deposited_amounts reducing the total balance of the user accordingly, and returns the status of payment to the proxy server.

5.8. COMPLETE REQUEST

The proxy server after receiving the status of the payment, if it was successful, it sends the device the reply that contains the results of the initial request through the initial socket connection, the result could be encrypted with the private key of the device for an extra level of security. Otherwise, it informs the device that the request could not be fulfilled.

6. SECURITY ANALYSIS

The application works end-to-end, and the communications get handled automatically. It would be interesting to investigate certain scenarios that would test the level of security and robustness of the architecture.

i. Considering there exists a malicious actor that tries to impersonate the device of a user knowing its id, in order to use the service at the expense of someone else. The system, as mentioned before uses RSA asymmetric keys to verify the identity of the device before charging for the service and handing back the results. This means, that in order for the malicious actor to impersonate the device, he has to build the correct signature. That in turn, means that he has to have knowledge of the private key of that device. Since the system never owns or processes that key, its security is out of the scope of the application and the system can be considered secure against this type of attack.

ii. Considering there is a malicious actor listening to the socket connection of the proxy server and the device and he alters the device request to fill his own needs. Even though the request will get handled by the proxy server, the user is in a place to get knowledge of the attack and have proof, by cross-checking using Etherscan and the oracle contract. The oracle contract holds information about every transaction handled through the service, so the attack could not get buried. Also, the mechanism that could handle a refund is already in place. What's more is that even if the actor succeeded, he would not be in a place to use the result without knowledge of the device's private key, so the attack could be pointless.

iii. In the case that multiple proxies have been employed, as supported by the architecture, the system is also protected by DDOS attacks. Every proxy would have the same amount of information, and requests would be equally distributed so when a single proxy is down, the system would be fully functional and there would be no loss of data or requests.

iv. In the scenario that the proxy server account gets compromised and wants to charge users without completing requests to make a profit. If the smart contract calls for some reason did not complete as expected, then the Etherscan page for the transaction hash would hold a status that would reflect the failure. So, in the management contract when checking the fields of the oracle, the verification would also fail, the user would not get charged, the result would not be sent back, and the user would be notified.

7. DISCUSSION

A demo of the proposed architecture that works end-to-end enabling devices to transact with any contract within an ethereum network has been implemented. However, this implementation is just a sample compared to the scale that could be achieved based on the system design. This does not mean that limitations don't exist. The architecture is dependent on outside sources that are considered to be trusted. But in case of their failure, the system cannot be protected. Furthermore, the line between data encryption and integrity is thin. Due to the computational limitations of ethereum, complex encryption can not be applied on a contract level, instead better security would be achieved on the proxy server. And yet, smart contracts due to the fact that their code is accessible to everyone, and their data cannot be altered, provide better clarity of the process and the state of information. In future works a demo using multiple proxies could be implemented, better encryption algorithms and security mechanisms could be explored and the matter of undependability from other resources to be researched.

REFERENCES

[1] S. Nakamoto, "A Peer-to-Peer Electronic Cash System", *Bitcoin.org*, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] Rodeck and J. Schmidt, "What Is Blockchain?", *Forbes.com*, 2022. [Online]. Available: https://www.forbes.com/advisor/investing/what-is-blockchain/

[3] "Distributed Ledger Technologies – SCETA", *Sceta.io*, 2022. [Online]. Available: https://sceta.io/distributed-ledger-technologies/

[4] "A Beginner's Guide to Understanding the Blockchain (Part 1: Introduction to Blockchain Technology)", *Medium*, 2022. [Online]. Available: https://medium.com/coinmonks/a-beginners-guide-to-investing-in-crypto-74781455645/

[5] S. Aggarwal and N. Kumar, "Cryptographic consensus mechanisms", *Advances in Computers*, pp. 211-226, 2021. Available: 10.1016/bs.adcom.2020.08.011

[6] "Proof-of-work (PoW) | ethereum.org", *ethereum.org*, 2022. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/

[7] "Ethereum accounts | ethereum.org", *ethereum.org*, 2022. [Online]. Available: https://ethereum.org/en/developers/docs/accounts/.

[8] "Merkle tree - Wikipedia", *En.wikipedia.org*, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Merkle_tree

[9]*Takenobu-hs.github.io*, 2022. [Online]. Available: https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

[10] D. Serpanos and T. Wolf, "Architecture of network systems overview", *Architecture of Network Systems*, pp. 1-9, 2011. Available: 10.1016/b978-0-12-374494-4.00001-3

[11] A. Barkalov, L. Titarenko and M. Mazurkiewicz, *Foundations of embedded systems*. p.28.

[12] Shurman, Mohammad & Obeidat, Abed & Al-Shurman, Saif. (2020). Blockchain and Smart Contract for IoT. 361-366. 10.1109/ICICS49469.2020.239551.

[13] S. Huh, S. Cho and S. Kim, "Managing IoT devices using blockchain platform," *2017 19th International Conference on Advanced Communication Technology (ICACT)*, 2017, pp. 464-467, doi: 10.23919/ICACT.2017.7890132.

[14] M. Samaniego, U. Jamsrandorj and R. Deters, "Blockchain as a Service for IoT," *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2016, pp. 433-436, doi: 10.1109/iThings-GreenCom-CPSCom-SmartData.2016.102.

[15] A. Dorri, S. S. Kanhere, R. Jurdak and P. Gauravaram, "Blockchain for IoT security and privacy: The case study of a smart home," *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2017, pp. 618-623, doi: 10.1109/PERCOMW.2017.7917634.

[16] Pavithran, D., Shaalan, K., Al-Karaki, J.N. *et al.* Towards building a blockchain framework for IoT. *Cluster Comput* **23,** 2089–2103 (2020). https://doi.org/10.1007/s10586-020-03059-5

[17]S. K. Lo *et al*., "Analysis of Blockchain Solutions for IoT: A Systematic Literature Review," in *IEEE Access*, vol. 7, pp. 58822-58835, 2019, doi: 10.1109/ACCESS.2019.2914675.

[18] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," in *IEEE Access*, vol. 4, pp. 2292-2303, 2016, doi: 10.1109/ACCESS.2016.2566339.

[19] Ge, C., Ma, X., & Liu, Z. (2020). A semi-autonomous distributed blockchain-based framework for UAVs system. *Journal Of Systems Architecture*, *107*, 101728. https://doi.org/10.1016/j.sysarc.2020.101728

CODE

I. EXEMPLARY SMART CONTRACT

```
pragma solidity ^0.5.0;
contract Test1 {

    struct Id{
      string name;
      uint am;
    }

    Id;

    function set_id( string memory _name, uint  _am) public {
        id.name=_name;
        id.am=_am;
    }

    function get_id() public view returns ( string memory, uint) {
        return(id.name, id.am);
    }
}
```

II. ORACLE SMART CONTRACT

```
pragma solidity ^0.8.0;


contract Oracle{

  struct Receipt{
    string tx_hash;
    string from;
    string to;
    string value;
    string tx_fee;
    string gas_limit;
    string gas_used;
    string method_id;
    string arguments;
  }
```

```
  bytes tempNum;
  string[] args;

  mapping (string => Receipt) public transactions;

  function set_data2(string memory _tx_hash, string memory _from,
string memory _to, string memory _value, string memory _tx_fee, string
memory _gas_limit, string memory _gas_used, string memory _method_id,
string memory _arguments) public{
    Receipt memory r;
    r.tx_hash = _tx_hash;
    r.from = _from;
    r.to = _to;
    r.value = _value;
    r.tx_fee = _tx_fee;
    r.gas_limit = _gas_limit;
    r.gas_used = _gas_used;
    r.method_id = _method_id;
    r.arguments =_arguments;
    transactions[_tx_hash] = r;
    delete _arguments;
  }

  function get_data(string memory _tx) public view returns(string
memory, string memory, string memory){
    Receipt memory rc = transactions[_tx];
    return(rc. tx_fee, rc.method_id, rc.arguments);
  }

}
```

III. MANAGEMENT SMART CONTRACT

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Management{

  address proxy = 0x14DC00aB538480A5512625cBd3D92Ee03fff15fF;
  address payable owner = payable(proxy);
  address oracle_a = 0x386EC2CFF43e9bd5A68310af975EB2FD435B98A4;
  bytes tempNum;
  string[] args;
  Oracle o = Oracle(oracle_a);
  bool checker;
```

```
  bool checkargs;
  bool checkmethod;
  mapping (address => uint) deposited_amounts;
  mapping (uint => address) device_ids;

  function str2num(string memory numString) public pure returns(uint) {
        uint  val=0;
        bytes   memory stringBytes = bytes(numString);
        for (uint  i =  0; i<stringBytes.length; i++) {
            uint exp = stringBytes.length - i;
            bytes1 ival = stringBytes[i];
            uint8 uval = uint8(ival);
           uint jval = uval - uint(0x30);

           val +=  (uint(jval) * (10**(exp-1)));
        }
      return val;
    }

  function bytes2num(bytes memory numString) public pure returns(uint)
{
        uint  val=0;
        for (uint  i =  0; i<numString.length; i++) {
            uint exp = numString.length - i;
            bytes1 ival = numString[i];
            uint8 uval = uint8(ival);
           uint jval = uval - uint(0x30);

           val +=  (uint(jval) * (10**(exp-1)));
        }
      return val;
    }

  function deposit(uint _id) external payable{
    require(msg.value > 0);
    deposited_amounts[msg.sender] = deposited_amounts[msg.sender] +
msg.value;
    device_ids[_id] = msg.sender;
  }

  function splitStr(string memory str, string memory delimiter) public
returns (string[] memory){
       delete args;
       bytes memory b = bytes(str);
       bytes memory delm = bytes(delimiter);
```

```
        for(uint i; i<b.length ; i++){

            if(b[i] != delm[0]) {
                tempNum.push(b[i]);
            }
            else {
                args.push(string(tempNum));
                tempNum = "";
            }
        }

        if(b[b.length-1] != delm[0]) {
           args.push(string(tempNum));
        }
        return args;

    }

    struct Request{
      string device_id;
      string contract_address;
      string funcname;
      string method_id;
    }

   struct Signature{
     bytes32 message;
     bytes s;
     bytes e;
     bytes m;
     address payable ver;
   }

   function request_payment(string memory request, string memory txhash,
string memory request_method_id, uint numofargs) external payable
returns(uint){
     Request memory r;
     Signature memory s;
     string[] memory request_data;
     string[] memory oracle_args;
     uint check=1;
     uint amount;
     request_data = splitStr(request, '#');
     r.device_id=request_data[0];
     //uint dev_id = str2num(request_data[0]);
     address user = device_ids[str2num(request_data[0])];
```

```
    r.contract_address= request_data[1];
    r.funcname= request_data[2];
    r.method_id = request_method_id;
    s.message = sha256(bytes(request_data[0]));
    s.s = bytes(request_data[(3 + numofargs)]);
    s.e = bytes(request_data[(4 + numofargs)]);
    s.m = bytes(request_data[(5 + numofargs)]);
    s.ver = payable(0xF21891F4Eae85a84C4Ff3a3c862E77FE08B85f42);
    (string memory tx_fee, string memory oracle_method_id, string
memory oracle_arguments) = o.get_data(txhash);
    oracle_args = splitStr(oracle_arguments, ' ');
    for(uint i = 3; i<request_data.length; i++){
      for(uint j =0; j<oracle_args.length; j++){
        if(keccak256(abi.encodePacked((request_data[i]))) ==
keccak256(abi.encodePacked((oracle_args[j])))){
          check=1*check;
        }
      }
    }
    if(keccak256(abi.encodePacked((r.method_id))) ==
keccak256(abi.encodePacked((oracle_method_id)))){
     checkmethod = true;
    }
    else{
      checkmethod = false;
    }
    if(check ==1){
      checkargs=true;
    }
    else{
      checkargs=false;
    }
    bool check_sig;

    bytes memory output_ver = check_verification(s);
    if(bytes2num(output_ver) == 0){
      check_sig = true;
    }
    else{
      check_sig = false;
    }
    checker = checkmethod && checkargs && check_sig;
    if(checker == true){
      amount = str2num(tx_fee);
      if(deposited_amounts[user]>=amount){
        (bool success, ) = owner.call{value: amount}("");
```

```
        require(success, "Failed to send Ether");
        deposited_amounts[user] = deposited_amounts[user] - amount;
      }
    }
    return(deposited_amounts[user]);
  }

  function get_checker() public view returns(bool){
    return checker;
  }

  function contract_balance() public view returns(uint256){
    return address(this).balance;
  }

  function get_balance(address _user) public view returns(uint256){
    return address(_user).balance;
  }

  function get_deposit_by_user(address user) public view
returns(uint256){
    return deposited_amounts[user];
  }

  function check_verification(Signature memory s) public payable
returns(bytes memory){
      bytes32 _sha256 = s.message;
      bytes  memory _s = s.s;
      bytes memory _e = s.e;
      bytes memory _m = s.m;
      address payable _addr = s.ver;
      bytes memory payload =
abi.encodeWithSignature("pkcs1Sha256Verify(bytes32,bytes,bytes,bytes)",
_sha256,_s,_e,_m);
      (bool success, bytes memory data) = _addr.delegatecall(payload);
      require(success);
      return data;
    }
}


interface Oracle{
    function set_data2(string memory _tx_hash, string memory _from,
string memory _to, string memory _value, string memory _tx_fee, string
memory _gas_limit, string memory _gas_used, string memory _method_id,
string memory _arguments) external;
```

```
    function get_data(string memory _tx) external view returns(string
memory, string memory, string memory);

}
```

IV. DATABASE MAKER

```
import sqlite3
from sqlite3 import Error


def create_connection(db_file):
    """ create a database connection to the SQLite database
        specified by db_file
    :param db_file: database file
    :return: Connection object or None
    """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        return conn
    except Error as e:
        print(e)

    return conn


def create_table(conn, create_table_sql):
    """ create a table from the create_table_sql statement
    :param conn: Connection object
    :param create_table_sql: a CREATE TABLE statement
    :return:
    """
    try:
        c = conn.cursor()
        c.execute(create_table_sql)
    except Error as e:
        print(e)


def main():
    database = r"users.db"

    sql_create_users_table = """ CREATE TABLE IF NOT EXISTS users (
                                        user_id text PRIMARY KEY,
```

```
                                        password text NOT NULL,
                                        account_address text NOT NULL,
                                        pk text NOT NULL
                                    ); """

    sql_create_devices_table = """CREATE TABLE IF NOT EXISTS devices (
                                        device_id integer PRIMARY KEY,
                                        user_id text NOT NULL,
                                        FOREIGN KEY (user_id) REFERENCES
users (user_id)
                                    );"""

    # create a database connection
    conn = create_connection(database)

    # create tables
    if conn is not None:
        # create projects table
        create_table(conn, sql_create_users_table)

        # create tasks table
        create_table(conn, sql_create_devices_table)
    else:
        print("Error! cannot create the database connection.")


if __name__ == '__main__':
    main()
```

V. PROXY SERVER

```
from web3 import Web3, HTTPProvider
import json
import sqlite3
from sqlite3 import Error
import usocket
import time
from hexbytes import HexBytes
import itertools, sys
spinner = itertools.cycle(['-', '/', '|', '\\'])
import os
```

```
account_address_str = os.getenv('ACCOUNT_ADDRESS')
management_address_str = os.getenv('MANAGEMENT_ADDRESS')
pk_str = os.getenv('PK')


def format_address(add):
    """
    Function that correctly formats an address for web3 usage
    :param add: The address before
    :return: The address formatted
    """
    before = "".join(add)
    address = Web3.toChecksumAddress(before.lower())
    return address


def format_key(pk):
    """
    Function that correctly formats a private key for web3 usage
    :param pk: The private key before
    :return: The private key formatted
    """
    before = "".join(pk)
    private_key = bytes.fromhex(before)
    return private_key



# creates a connection to the database

def create_connection(db_file):
    """ create a database connection to the SQLite database
        specified by db_file
    :param db_file: database file
    :return: Connection object or None
    """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
    except Error as e:
        print(e)

    return conn


def query_info(conn, device_id):
    """
```

```
    Retrieves account information from database
    :param conn: the connection to the database
    :param device_id: The id of the device
    :return: the account information retrieved
    """
    cur = conn.cursor()
    query_id = "SELECT user_id FROM devices WHERE device_id =" +
str(device_id)
    cur.execute(query_id)
    user_id = "".join(cur.fetchone())
    query_account = "SELECT account_address FROM users WHERE user_id =
'" + user_id + "'"
    cur.execute(query_account)
    account_address_fromdb = cur.fetchone()
    account_address_before = "".join(account_address_fromdb)
    query_pk = "SELECT pk FROM users WHERE user_id ='" + user_id + "'"
    cur.execute(query_pk)
    private_key_before = "".join(cur.fetchone())
    return account_address_before, private_key_before


def extract_user_info(device_id):
    """
    Given a the id of user provides the account address and private key
correctly formatted
    :param device_id: The id of the device
    :return: The formatted information about the account and key
    """

    database = r"users.db"

    # create a database connection
    conn = create_connection(database)
    with conn:
        db_info = query_info(conn, device_id)
        account_address = Web3.toChecksumAddress(db_info[0].lower())
        priv_key = bytes.fromhex(db_info[1])
        info = (account_address, priv_key)
    return info


def spinning_cursor():
    while True:
        for cursor in '|/-\\':
            yield cursor
```

```
def send_txhash(txhash):
    """
    This method recieves a transaction hash and using web sockets sends
it to the third-party
    :param txhash: The hash of the transaction sent
    :return: None
    """
    with usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM) as s:
        s.connect(('127.0.0.1', 6533))
        s.send(txhash)
        s.close()


def checking_nonce():
    """
    This function retrieves information about the nonce of the account
the server uses to transact
    :return: The nonce retrieved from web3
    """
    w3_ =
Web3(HTTPProvider("https://ropsten.infura.io/v3/cbe831fc4e9a47699e55af9
141b7930f"))
    nonce =
w3_.eth.get_transaction_count('0x14DC00aB538480A5512625cBd3D92Ee03fff15
fF')
    return nonce


def contract1(contract_address, priv_key, account_address, func_name,
args, abi):
    """
    This function sends a raw transaction to a contract or makes a call
to a contract depending on the parameters
    :param contract_address: The address of the contract to interact
with
    :param priv_key: The private key of the account to make the
transaction
    :param account_address: The address of the account to make the
transaction
    :param func_name: The name of the function to call/interact with
    :param args: A list of the arguments to be used
    :param abi: The abi of the contract
    :return: The output of the call, the hash of the transaction
performed, the hash of the transaction in bytes and
     the nonce of the account before making the transaction
    """
```

```
    w3_ =
Web3(HTTPProvider("https://ropsten.infura.io/v3/cbe831fc4e9a47699e55af9
141b7930f"))
    entity = w3_.eth.contract(address=contract_address, abi=abi)
    func = getattr(entity.functions, func_name)
    time.sleep(10)
    nonce = w3_.eth.get_transaction_count(account_address)
    if func_name =="request_payment":
        value = w3_.toWei('100', 'gwei')
    else:
        value = 0
    print(f"Trying to transact with {func_name} and nonce is {nonce}")
    entity_txn = func(*args).buildTransaction({
        'from': account_address,
        'chainId': 3,
        'value': value,
        'gas': 3000000,
        'gasPrice': w3_.toWei('200', 'gwei'),
        'nonce': nonce,
    })
    signed_txn = w3_.eth.account.sign_transaction(entity_txn,
private_key=priv_key)
    tx_hash_b = w3_.eth.send_raw_transaction(signed_txn.rawTransaction)
    if func_name == 'request_payment':
        output = []
    else:
        output = func(*args).call()
    while w3_.eth.get_transaction_count(account_address) == nonce:
        sys.stdout.write(next(spinner))  # write the next character
        sys.stdout.flush()  # flush stdout buffer (actual character
display)
        sys.stdout.write('\b')  # erase the last written char
        #time.sleep(2)
    print(f"Out of loop and nonce is
{w3_.eth.get_transaction_count(account_address)}")
    tx_hash = bytes(HexBytes(tx_hash_b)).hex()
    return [output, tx_hash, tx_hash_b, nonce]


# takes the data received from Pycom and formats them
# data[0] -> device id, data[1] -> private key for end to end
encryption

def decode_request(request):
    """
```

```
    This function formats device requests, fulfils them, and informs
the third-party about the transaction hash
    :param request: The received request
    :return: The formatted reply to the device with the outcome, the
transaction hash, the id of the method called,
     and the nonce of the account when the transaction was made
    """
    reply = bytearray()
    args = []
    priv_key = format_key(pk_str)
    account_address = format_address(account_address_str)
    data = request.decode().split("#")
    print(data)
    num_of_args = len(data)
    num_of_method_args = num_of_args - 6
    print('length of data is ' + str(num_of_args))
    device_id_b = data[0].encode('utf-8')
    device_id = device_id_b.decode('utf-8')
    contract_address_before_b = data[1].encode('utf-8')
    contract_address_before = contract_address_before_b.decode('utf-8')
    func_name_b = data[2].encode('utf-8')
    func_name = func_name_b.decode('utf-8')

    pos = "build\contracts" + chr(92) + contract_address_before +
".json"
    with open(pos) as f:
        info_json = json.load(f)
    abi = info_json["abi"]
    types = []
    index = 0
    str_for_id = func_name + '('
    int_types = {'int256', 'uint256'}
    for i in info_json["abi"]:
        for j in i.get('inputs'):
            types.insert(index, j.get('type'))
            if index != ((num_of_args-3) - 4):
                str_for_id = str_for_id + types[index] + ','
            else:
                str_for_id = str_for_id + types[index] + ')'
            index += 1
    for i in range(3, num_of_args-3):
        if types[i - 3] in int_types:
            args.insert(i - 3, int(data[i]))
        else:
            args.insert(i - 3, data[i])
    print(args)
```

```
    print(str_for_id)
    w3 = Web3()
    method_id = w3.sha3(text=str_for_id)[0:4].hex()
    print(method_id)
    account_address = extract_user_info(device_id)[0]
    priv_key = extract_user_info(device_id)[1]
    contract_address =
Web3.toChecksumAddress(contract_address_before.lower())
    result = contract1(contract_address, priv_key, account_address,
func_name, args, abi)
    send_txhash(result[2])
    reply.extend(
        b'The function called was ' + func_name.encode('utf-8') + b'
and the result was ' + bytes(str(result[0]),

                             'utf-8'))
    return [reply, result[1], method_id, result[3], num_of_method_args]



# Set up server socket



def main():
    management_address = format_address(management_address_str)
    priv_key = format_key(pk_str)
    serversocket = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM)
    serversocket.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR,
1)
    serversocket.bind(("", 6544))

    # Accept maximum of 5 connections at the same time
    print("Listening..")
    serversocket.listen(2)

    # Unique data to send back

    while True:
        # Accept the connection of the clients
        print("accepting..")
        (clientsocket, address) = serversocket.accept()
        print('Connection from', address)
        request = clientsocket.recv(1000)
        print('Got %s', request)
        returned = decode_request(request)
        reply = returned[0]
```

```
        tx = "0x" + returned[1]
        request_data = request.decode()
        pos = "build\contracts" + chr(92) + "Management.json"
        with open(pos) as f:
            info_json = json.load(f)
        abi = info_json["abi"]
        print(request_data + ' ' + tx)
        method_id = returned[2].split('0x')[1]
        print(method_id)
        first_nonce = returned[3]
        num_of_args = returned[4]
        while checking_nonce() != first_nonce+2:
            print(f'Checking nonce... {str(checking_nonce()}')
            time.sleep(2)
        status = contract1(management_address, priv_key,
account_address, 'request_payment', [request_data, tx, method_id,
num_of_args], abi)
        if status == True:
            print("sending reply...")
            print(reply)
            clientsocket.sendall(reply)
        else:
            clientsocket.sendall(b'Error in request payment.')
        clientsocket.close()
```

VI. THIRD PARTY

```
import requests
import json
from bs4 import BeautifulSoup
import xmltojson
import usocket
from hexbytes import HexBytes
import time
from web3 import Web3, HTTPProvider
from textwrap import wrap
import os

tag_dict = ['spanTxHash', 'spanFromAdd', 'spanToAdd',
'ContentPlaceHolder1_spanValue', 'ContentPlaceHolder1_spanTxFee',
            'ContentPlaceHolder1_spanGasLimit',
'ContentPlaceHolder1_spanGasUsedByTxn', 'inputdata']

header = {
```

```
    "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169
Safari/537.36",
    'referer': 'https://www.google.com/'
}

account_address_str = os.getenv('ACCOUNT_ADDRESS')
management_address_str = os.getenv('ORACLE_ADDRESS')
pk_str = os.getenv('PK')



def get_txhash():
    """
    Function that receives the hash of a transaction from the proxy
server
    :return: The transaction hash received
    """
    ssocket = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM)
    ssocket.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR, 1)
    ssocket.bind(('', 6533))
    print("Listening to proxy server...")
    ssocket.listen(1)
    print("Accepting..")
    (csocket, address) = ssocket.accept()
    btxhash = csocket.recv(1000)
    ssocket.close()
    print('Got %s', btxhash)
    txhash = bytes(HexBytes(btxhash)).hex()
    return txhash


def format_address(add):
    """
    Function that correctly formats an address for web3 usage
    :param add: The address before
    :return: The address formatted
    """
    before = "".join(add)
    address = Web3.toChecksumAddress(before.lower())
    return address


def format_key(pk):
    """
    Function that correctly formats a private key for web3 usage
    :param pk: The private key before
```

```
    :return: The private key formatted
    """
    before = "".join(pk)
    private_key = bytes.fromhex(before)
    return private_key



def oracle_data(contract_address, priv_key, account_address, func_name,
args, abi):
    w3_ =
Web3(HTTPProvider("https://ropsten.infura.io/v3/cbe831fc4e9a47699e55af9
141b7930f"))
    entity = w3_.eth.contract(address=contract_address, abi=abi)
    func = getattr(entity.functions, func_name)
    nonce = w3_.eth.get_transaction_count(account_address)
    print('before transaction the nonce is: ' + str(nonce))
    entity_txn = func(*args).buildTransaction({
        'from': account_address,
        'chainId': 3,
        'gas': 3000000,
        'gasPrice': w3_.toWei('200', 'gwei'),
        'nonce': nonce,
    })
    signed_txn = w3_.eth.account.sign_transaction(entity_txn,
private_key=priv_key)
    tx_hash = w3_.eth.send_raw_transaction(signed_txn.rawTransaction)
    while w3_.eth.get_transaction_count(account_address) == nonce:
        time.sleep(1)
    return tx_hash


def formatting_input(inputdata):
    arguments = ""
    if inputdata[0] == 'F':
        method_name = inputdata.split()[1]
        w3 = Web3()
        method_id = w3.sha3(text=method_name)[0:4].hex()
        return method_id, arguments
    else:
        method_id = (inputdata.split('0x')[1]).split('0')[0]
        rest = (inputdata.split('0x')[1]).split(method_id)[1]
        tobesorted = wrap(rest, 64)
        times = len(tobesorted)
        print(tobesorted)
        for i in range(1, times, 2):
```

```
            if tobesorted[i][0] == '0':
                decoded1 =
int.from_bytes((bytes.fromhex(tobesorted[i])), byteorder='big')
            else:
                decoded1 =
(bytes.fromhex(tobesorted[i].split('0')[0])).decode('utf-8')
            arguments = arguments + str(decoded1) + ' '
        return method_id, arguments



def main():
    account_address = format_address(account_address_str)
    contract_address = format_address(contract_address_str)
    priv_key = format_key()
    pos = "build\contracts" + chr(92) + "Oracle.json"
    with open(pos) as f:
        info_json = json.load(f)
    abi = info_json["abi"]
    func_name = "set_data2"
    func_name_get = "get_data"
    tx = get_txhash()
    print(tx)
    tx_url_front = "https://ropsten.etherscan.io/tx/0x"
    tx_url = tx_url_front + tx
    print(tx_url)
    time.sleep(15)
    status = 0
    while status != 1:
        r_status = requests.get(url=tx_url, headers=header)

        # checking if status is pending, if it is the status class will
be col-md-9 and if successful it'll be col col-md 9

        soup = BeautifulSoup(r_status.text, "html.parser")
        status_line = soup.find("div", {"class": "col col-md-9"})  # if
this cannot be found then status is pending
        print(status_line)
        if status_line is None:
            status = 0
            time.sleep(5)
        else:
            status = 1

    # status is success
    r = requests.get(url=tx_url,
```

```
                    headers=header)
    jsonlist = []
    html_doc = r.text
    soup = BeautifulSoup(r.text, "html.parser")
    for _id in tag_dict:
        wline = soup.find(id=_id)
        print(_id)
        print(wline)

        with open("sample2.html", "w") as html_file2:
            html_file2.write(str(wline))

        with open("sample2.html", "r") as html_file2:
            html = html_file2.read()
            json_ = xmltojson.parse(html)

        with open("data.json", "w") as file:
            json.dump(json_, file)
            jsonlist.append(json_)

    names = ['Transaction Hash', 'From', 'To', 'Value', 'Transaction
Fee', 'Gas Limit', 'Gas Used by Transaction']
    print(jsonlist)
    data = {}
    y = 0
    for x in names:
        mdict = json.loads(jsonlist[y])
        ndict = mdict['span']
        print(x)
        print(y)
        try:
            data[x] = (ndict['#text'])
        except:

            try:
                print("entered except")
                nndict = ndict['span']
                data[x] = (nndict['#text'])
            except:
                nndict = ndict['i']
                print("Transaction is still pending...")
                status = 1
        y += 1
    mdict = json.loads(jsonlist[y])
    ndict = mdict['textarea']
    # data['Data'] = (ndict['#text'])
```

```
    inputdata = (ndict['#text'])
    formatted_input_data = formatting_input(inputdata) #
formatted_input_data[0] will be the method_id and [1] the arguments
    data['Method_id'] = formatted_input_data[0]
    print(formatted_input_data[1])


    data['Arguments'] = formatted_input_data[1]
    print(data['Arguments'])
    print(type(data['Arguments']))

    print(data)
    data['Value'] = data['Value'].split('($')[1]
    data['Value'] = data['Value'].split(')')[0]
    data['Transaction Fee'] = data['Transaction Fee'].split(' Ether
($0.00)')[0]
    data['Transaction Fee'] = data['Transaction Fee'].split('0')[1]
    dummy = data['Transaction Fee']
    num = 0
    for i in range(0, len(dummy)):
        if dummy[i] == '0':
            num += 1
        else:
            break
    data['Transaction Fee'] = data['Transaction Fee'][num-1:
len(dummy)-1]
    args = []

    for x in data:
        args.append(data[x])
    args_get =
['0x53c2db9715da55d34a1db5be86afcc440c9827649cb163db1a2e366f66d4fc36']

    print((oracle_data(contract_address, priv_key, account_address,
func_name, args, abi)).hex())
```

VII. EXEMPLARY DEVICE CODE

```
import socket
import _thread
import time
import uerrno
```

```
host = '192.168.1.23'
port = 6544

contract_test_1 = "0x3cEb3729F84493ca0a8282552a107c99BF1AEfCB"
contract_test_2 = "0xd3179E7FCc8bbf0c4065d33B958Efd55fe7D5Bd8"
print('Creating socket..')

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to remote server
print('# Connecting to server')
s.connect((host , port))
# Send data to remote server
print('# Data encoded for the request: device id, contract_address,
function name, parameters, signature')
request =
"7#0x3cEb3729F84493ca0a8282552a107c99BF1AEfCB#gset_id#maria#1417#0x5B38
Da6a701c568545dCfcB03FcB875f56beddC4"

s.sendall(request)

# Receive data
print('# Receive data from server')
reply = s.recv(1000)
reply_decoded = reply.decode('utf-8')
print(reply_decoded)

s.close()
```