POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

CrowdPatching: Decentralized Distribution of IoT Software Updates

Design, Security Analysis and Implementation of a Decentralized and Scalable Protocol for IoT Software Updates Delivery, Leveraging Self-Interested Distributors, Blockchain Smart Contracts and Zero-Knowledge Proofs

Supervisors
Prof. Arash Shaghaghi
Prof. Antonio Lioy

Candidate
Edoardo Puggioni

Academic Year 2019-2020

To my family

Summary

The number of Internet of Things (IoT) devices deployed around the world is growing at incredible speed. The primary goal of their design is optimizing their size, cost and usability, while their security is underestimated. As a consequence, they often present serious vulnerabilities, posing security threats to both individual users and organizations. For this reason, delivering software updates to these devices to patch their vulnerabilities is crucial. In this context, manufacturers face three main challenges. Firstly, the integrity of new updates must be strictly enforced to avoid the installation of malicious software, which would create more threats than it mitigates. Secondly, only efficient and lightweight protocols can be adopted, to account for the limited hardware and software resources characterizing IoT smart objects. And thirdly, one crucial issue is scalability: software patches are currently delivered by means of traditional client-server architectures, which is not a sustainable approach considering the number of devices involved. Motived by these limitations, we propose *CrowdPatching*, a decentralized protocol leveraging blockchain technologies and zero-knowledge proofs, where IoT manufacturers delegate the delivery of software updates to self-interested distributors in exchange for cryptocurrency. Manufacturers announce new updates by deploying a smart contract, which in turn will issue cryptocurrency payments to any distributor who provides an unforgeable proof-of-delivery. The latter consists in a signature provided by IoT devices when they receive a valid zero-knowledge proof from the distributor.

Compared with related work, the *CrowdPatching* protocol offers three main advantages. First, the number of distributors can scale indefinitely. The update is initially shared by the manufacturer with a finite set of distributors. Other proposals do not allow this set to grow at a later time. Instead, we introduce a mechanism through which distributors can share the update with others in exchange for a cryptocurrency payment. Furthermore, we leverage the recent common integration of Hub (or gateway) devices in IoT deployments, by letting them perform the most demanding actions of the protocol. As a consequence, the protocol is feasible even for the more constraint IoT objets. Finally, we propose a score system for distributors, which records their trustworthiness on the blockchain and rewards honest behavior. We provide an informal security analysis of the *CrowdPatching* protocol, analyzing possible attacks, as well as the corresponding protections and mitigations. And we also provide a formal security analysis, which was performed by means of the Tamarin Prover, a state-of-the-art protocol analysis tool allowing to verify security properties in the symbolic model. What is more, we present a prototype implementation, enabling the execution of all protocol steps. In particular, we focus on the implementation of (*i*) the blockchain smart contracts and (*ii*) the zero-knowledge proving system. The former is based on Ethereum, the second most popular blockchain platform after Bitcoin. While the latter is based on the zk-SNARKs proving system, and exploits the most advanced cryptographic library available in this context, called libsnark.

Acknowledgements

I would first like to thank my supervisor at Deakin University, Prof. Arash Shaghaghi. This thesis work would not have been possible without his constant guidance and support. His expertise and leadership were inspiring, allowing me to grow in both academic and personal terms.

In addition, I would like to thank Prof. Robin Doss from Deakin University, who often dedicated his precious time to help the making of this project, providing essential advice and ideas.

Furthermore, I would like to acknowledge the School of Information Technology at Deakin University, for hosting me during my research visit and providing all the necessary facilities.

I would also like to thank my supervisor at the Polytechnic University of Turin, Prof. Antonio Lioy, who provided rigorous feedback and insightful suggestions throughout this work.

Finally, I would like to thank my family, my close relatives and my friends, for their fundamental support throughout this journey. Special thanks must go to my parents, who truly supported me in all possible ways, including financially and emotionally.

Contents

1 Introduction 9

2 Background 11

2.1 Blockchain and Bitcoin . . . 11

2.2 Ethereum and Smart Contracts . . . 12

2.3 Zero Knowledge Proofs and zk-SNARKs . . . 13

3 IoT Software Updates 15

3.1 Challenges for IoT Software Updates . . . 15

3.2 Related Work . . . 17

3.3 Our Case Study . . . 19

3.3.1 Framework Entities . . . 20

3.3.2 Protocol Steps . . . 20

3.3.3 Discussion . . . 23

4 Proposed Protocol 25

4.1 Participating Entities . . . 25

4.1.1 Blockchain Platform . . . 25

4.1.2 Peer-to-peer file sharing . . . 26

4.1.3 Manufacturers and IoT Objects . . . 26

4.1.4 Hubs . . . 27

4.1.5 Distributors . . . 27

4.2 Protocol Steps . . . 28

4.2.1 Super Smart Contract . . . 28

4.2.2 Update Release . . . 28

4.2.3 Initial Seed and Additional Sharing . . . 30

4.2.4 Update Delivery . . . 31

4.2.5 Key Publication and Final Delivery . . . 32

5 Security Analysis 35
5.1 Informal Analysis 35
5.1.1 Impersonation Attacks 35
5.1.2 Interception of the PoD Submission 36
5.1.3 Malicious Distributors 37
5.1.4 Update Integrity 38
5.1.5 Old Update Delivery 38
5.2 Formal Analysis 38
5.2.1 The Tamarin Prover 38
5.2.2 *CrowdPatching* Protocol Rules 42
5.2.3 *CrowdPatching* Security Properties 50

6 Implementation 53
6.1 Ethereum Smart Contracts 53
6.1.1 Solidity 53
6.1.2 Super Smart Contract (SSC) 53
6.1.3 Delivery Smart Contract (DSC) 57
6.1.4 Exchange Smart Contract (ESC) 61
6.1.5 Deployment on the Blockchain 62
6.2 zk-SNARKs Proving System 62
6.2.1 The libsnark library 62
6.2.2 The jsnark library 63
6.2.3 Our proving system 63
6.3 Additional Software 75
6.3.1 Digital Signatures 75
6.3.2 Block Cipher in CBC Mode 75

7 Conclusions and Future Work 76

Bibliography 77

A Solidity Developer Manual 79
A.1 Installation Instructions 79
A.2 Build and Deploy Smart Contracts 79
A.2.1 Run Ganache (Ethereum Local Blockchain) 80
A.2.2 Configure Truffle and Ganache 80
A.2.3 Deploy the SSC smart contract 80
A.2.4 Interact with the SCC to deploy DSCs and ESCs (and more) 80

B zk-SNARKs User Manual 81

B.1 Prerequisites 81

B.2 Compiling Instructions 81

B.3 Execute the zk-SNARKs Algorithms 82

B.3.1 *Setup*: Generate the Proving and Verifying Keys 82

B.3.2 *Prove*: Generate the Proof File 82

B.3.3 *Verify*: Verify the Proof File 83

Chapter 1

Introduction

The number of Internet of Things (IoT) devices deployed worldwide is growing at an incredible speed, projected to reach more than 75 billion by 2025 [1]. Security is often an afterthought for the manufacturers of these devices, with their primary concern being cost, size, and usability. Hence, IoT devices often present vulnerabilities that attackers can exploit, posing serious threats to organizations and individuals in terms of security and privacy. As a consequence, regular software updates are fundamental in securing vulnerable IoT devices. In this context, manufacturers face a number of key challenges [2, 3, 4, 5]. One of the most important requirements is integrity of the updates. It is fundamental that updates are not tampered with, otherwise the patching process would create more threats than it mitigates, leading to malicious code being executed. Secondly, IoT objects are usually characterized by restricted hardware and software resources. Hence, the need for efficient and lightweight cryptographic primitives and protocols. Thirdly, given the number of devices involved, another crucial aspect is scalability. Manufacturers currently deliver updates by means of traditional client-server architectures. However, this approach is clearly not sustainable when we consider the number of devices, different versions of the same devices, iteration of updates and requirement to support legacy devices.

In this thesis, we propose *CrowdPatching*, a protocol allowing manufacturers to deliver updates to IoT devices in a decentralized manner, leveraging blockchain technologies and zero-knowledge proofs. In this system, the manufacturer delegates the delivery of new updates to self-interested agents, called distributors, who undertake this task in exchange for cryptocurrency payments. First of all, we assume to have a permissionless blockchain as an underlying infrastructure, natively supporting cryptocurrency and smart contracts. Manufacturers announce a new update release by deploying a smart contract, which in turn will issue cryptocurrency payments to any distributor who provides a proof-of-delivery. The latter is a signature generated by an IoT device. It works as an unforgeable digital commitment through which an IoT device is authorizing the smart contract to issue a payment to a certain distributor. For this reason, this important signature is produced by an IoT device only if specific conditions are met. In particular, a distributor is required to provide an encrypted version of the update, along with the hash value of the symmetric key employed. What is more, a distributor needs to produce evidence that this encrypted file was indeed derived from the official update, using a key with the provided hash value. In this context, the use of a zero-knowledge tool, called zk-SNARKs, is fundamentally important. Before a proof-of-delivery can be issued, a distributor is expected to provide a zk-SNARKs proof about the provided encrypted update, mathematically proving that (i) it was indeed obtained encrypting the update file authorized by the manufacturer and (ii) the employed key has indeed the provided hash value. Most importantly, this zero-knowledge proof does not reveal anything else beyond those facts: the unencrypted update and the key remain secret. If the proof is valid, then the proof-of-delivery signature is generated by the IoT device, and sent to the smart contract by the distributor along with the encryption key. Before issuing the payment, the contract checks that the signature is valid, and that the key matches the hash value. Along with the payment, the smart contract publishes the key on the blockchain. The latter can be now used to decrypt the file, and the update is finally obtained by the IoT device.

Compared to similar proposals [6, 7, 8, 9] discussed in Chapter 3, our design presents several

novelties and optimizations. In particular, we addressed a number of issues in the work of Leiba et al. [9], which constitutes the main inspiration for our proposal. First of all, our design allows the number of distributors to fully scale with the number of IoT devices. Indeed, any distributor who could not obtain the update file from the manufacturer, can acquire it from other distributors in exchange for a cryptocurrency payment. This is in contrast with the proposal of Leiba et al. where distributors can obtain the update file only in an initial phase, and new distributors willing to participate at a later moment cannot do so in any way. We argue that this limitation significantly hinders the scalability of their system, as it does not allow the number of distributors to grow indefinitely. Furthermore, we drastically reduce the requirements necessary for IoT objects to participate in the protocol, to account for their limited resources. We achieve this by means of a new participant, called hub, which functions as a trusted gateway for IoT devices in their local networks, performing the most demanding protocol steps in place of them. In particular, differently from related works, IoT devices are not required to participate in the blockchain network in any way, and can avoid verifying any zk-SNARKs proof. Finally, we introduce a simple mechanism allowing to judge the trustworthiness of distributors, leveraging a data structure automatically maintained by the smart contract. This mechanism is particularly important in a context where literally anyone can act as a distributor, and no other trust-related assumption can be made about these entities. What is more, we exploit the only reliable information a distributor is required to provide: the public key used as recipient for the cryptocurrency payments. We achieve this through a key-value database stored on the blockchain, associating each distributor's public key with the number of successful deliveries performed through time. In other words, when a certain distributor delivers an update file for the first time ever, a new entry is added to this database associating its public key with the integer value 1. After each subsequent successful delivery, this value is incremented. In this way, other participants of the protocol, especially hubs, can access the blockchain to read this value when choosing a distributor to interact with, selecting the one with the highest score. This mechanism encourages distributors to behave honestly, keeping a secure and unforgeable record of successful deliveries.

We produced an informal security analysis of the proposed protocol, discussing known attacks and evaluating possible vulnerabilities. The results suggest that the protocol is highly resistant to the most crucial threats, and presents well-balanced mitigations for other non-critical attack vectors. What is more, we performed a formal analysis by means of the Tamarin Prover, a state-of-the-art protocol analysis tool allowing to verify security properties in the symbolic model. We first developed a model representing all steps of the protocol, which must be defined as a collection of multiset rewriting rules. This included the execution of all the cryptographic functions involved. Additionally, since Tamarin does not have native support for the zero-knowledge proving system we employed, we had to provide custom definitions for the corresponding functions. We then demonstrated the executability of the protocol, which is to obtain mathematical assurance about the well-formedness of all its steps. Finally, we were able to prove three important security properties, defined as mathematical formulas over the rules composing the model. Indeed, the validity of these properties gave us tangible confidence about the security and fairness of the protocol. One property in particular allowed us to discover a serious vulnerability, capable of disrupting the security of the protocol and also applicable to a related research work. We were then able to fix the vulnerability so that the property was eventually satisfied.

Finally, we developed a prototype implementation of the *CrowdPatching* protocol, enabling the execution of all its steps. In particular, we focused on the implementation of (i) the smart contracts and (ii) the zero-knowledge proving system. The former is based on Ethereum, the second most popular blockchain platform after Bitcoin, and includes three smart contracts. While the latter is based on the zk-SNARKs proving system, a novel construction allowing to generate non-interactive proofs, and exploits an advanced cryptographic library called libsnark.

We start this thesis by providing the necessary background in Chapter 2. We continue in Chapter 3 by discussing the challenges associated with the issue of IoT software updates, and analyzing the corresponding related works. In Chapter 4, we illustrate the details of our proposal, discussing all protocols steps in depth. We then present both the informal and formal security analysis of the protocol in Chapter 5. Subsequently, we illustrate our prototype implementation in Chapter 6. Finally, we discuss conclusions and future work in Chapter 7.

Chapter 2

Background

The purpose of this chapter is to provide the necessary background to understand the proposed protocol, as well its analysis and the related research studies. We provide a brief overview of blockchain technologies in Section 2.1, and a presentation of its most famous implementation, Bitcoin. We continue describing another important instance in the evolution of blockchain in Section 2.2. That is, the Ethereum platform. Here we also discuss smart contracts, the main novelty introduced by Ethereum. Finally, Section 2.3 provides an overview of zero-knowledge proof systems, focusing on a specific zero-knowledge tool called zk-SNARKs.

2.1 Blockchain and Bitcoin

The term *blockchain* can be simply used to refer to a certain type of data structure. That is, an immutable chain, or sequence, of blocks. Each block within this structure contains (1) a payload and (2) the hash value of the previous block in the same chain. As a consequence of this configuration, the chain presents an important property: immutability. It is impossible for an attacker to tamper with any of the blocks constituting the chain. Any modification would break the chain of hash values, provoking a mismatch at some point in the chain. What is more, tampering with the blockchain becomes more difficult as the sequence grows longer. Indeed, an attacker intentioned to modify a transaction on a certain block would have to modify all subsequent blocks in the chain. However, the term *blockchain* most commonly refers to a much broader set of concepts, introduced along with Bitcoin in a white paper anonymously authored under the pseudonym of Satoshi Nakamoto [10]. Bitcoin, as described by the title of the paper, is a peer-to-peer electronic cash system where users can exchange a digital currency, often referred to as cryptocyrrency. In this context, the blockchain can be described as a distributed database storing a record of transactions in a permanent and tamper-proof manner. This record, often referred to as a public ledger, is fully decentralized: an entire copy of the database is maintained by each node participating in the peer-to-peer network.

The Bitcoin network allows two types of participants: passive nodes, which can only read from the blockchain, and active nodes, which can also write into the blockchain and are called *miners*. Miners play a fundamental role in the system, attaching new blocks to the blockchain. In practice, a miner performs the following steps. First of all, it gathers new transactions from other users, who broadcast them in the peer-to-peer network. Once the miner has received enough transactions to form a block, it can proceed verifying the validity of such transactions. This verification includes the analysis of previous transaction forming the current blockchain, which must be compatible with the new transactions. If the new transactions are valid, the miner continues initiating the consensus protocol. That is, a procedure allowing all participants of the peer-to-peer network to agree on the validity of the new block. This is done in a distributed fashion, without the need for a trusted third party. Different consensus protocols are employed in different cryptocurrency systems. In Bitcoin, the Proof-of-Work algorithm is used. Here, the miner must generate the solution for a mathematical puzzle based on the specific block. In general, these puzzles are very difficult to solve, but their solution is easy to verify. Once the solution is computed, the miner

attaches it to the new block, which can now be broadcasted to the network. Other peers will accept it as a valid block upon verification of the puzzle solution. If the majority of the peers of the Bitcoin network accept it, the block is added to the blockchain. As a consequence, all peers update their downloaded copy of the blockchain. An important detail hidden in the previous steps is the following. According to the Bitcoin protocol, a miner adds a specific transaction to the block, before computing the corresponding puzzle solution. That is, a transaction where new cryptocurrency is created and sent to the miner itself as a reward. This constitutes the only way in which new currency is injected into the network. The identities of participant nodes are managed through public key cryptography. Any account must posses a private key, which is kept secret, and the corresponding public key. What is more, transactions can be considered valid only if they are presented with a signature, generated by the sending party by means of its private key. In other words, if a transaction represents a cryptocurrency transfer from public key pub^A to pub^B, it must be accompanied by a signature made by A on the transaction itself, by means of its private key prv^A. A signature by the receiving end B is not required.

In general, blockchains can be permissionless or permissioned. In the first case, anyone is allowed to access the blockchain. More precisely, any node can read the blockchain or generate transactions, or even act as a miner, without any need for authentication. The Bitcoin blockchain falls in this category. On the other hand, permissioned blockchains can be accessed by authorized nodes only. Most importantly, only a selected number of actors can participate in the consensus protocol to approve the validity of new blocks.

2.2 Ethereum and Smart Contracts

After Bitcoin, new alternative blockchain systems were introduced. Many of these systems are analogous to Bitcoin in most of their features. However, others presented applications far beyond the mere exchange of digital cash among participant nodes. One of those applications are *smart contracts*. The term was coined by Szabo in a 1997 paper [11], long before the invention of Bitcoin. Here the author explains the idea through an example, where smart contracts are compared to vending machines. Any entity in possession of coins can interact with a vending machine, which will react automatically providing products (and change) according to the price tag. What is more, security mechanisms are in place, and cost of breaking them is higher than the benefits for a potential malicious user. However, the first practical applications of such concept were proposed years later, exploiting distributed ledgers and their consensus algorithms. In this context, smart contracts can be defined as computer code deployed on the blockchain, executed securely and automatically in a distributed fashion when certain conditions are met. Similarly to blockchain systems supporting solely cryptocurrency exchanges, the operation of a smart contract are triggered by valid transactions performed by nodes on the network, and the results are also recorded in the distributed ledger.

The most important platform with support for smart contracts is Ethereum, a blockchain system introduced by Buterin in 2014 [12], currently the second most widely adopted blockchain platform supporting cryptocurrency exchanges, after Bitcoin. Ethereum supports *externally owned* accounts, owned by users of the network, and smart contract accounts. External accounts are controlled by regular users holding private keys, while smart contract accounts are automatically managed by their own code. Users can create a new contract by sending a transaction to a specific fixed address, called the *zero-account*. Furthermore, they need to attach the code of the smart contract itself, as well as the needed parameters and an arbitrary amount of cryptocurrency. As a consequence of these action, the smart contract is deployed to the blockchain, and is represented by a new contract account. The parameters sent by the creator constitute the initial state of the contract, while the cryptocurrency forms its initial fund. From this point on, any user can send transactions to the address belonging to the contract account. This would trigger the execution of the contract code. Depending on the conditions, many actions can be performed, such us changing the state of the contract by writing in its storage on the blockchain, or automatically issuing transactions to other accounts, which in turn can be users or other other smart contracts. To avoid malicious actors exploiting the automatic nature of code execution, all Ethereum operations requiring computational effort have an associated fee, which is measured with a special unit

called *gas*. As in Bitcoin, users can also exchange cryptocurrency between each others, without interacting with smart contracts at all.

2.3 Zero Knowledge Proofs and zk-SNARKs

A zero knowledge proof is a cryptographic tool allowing one party, the *prover*, to convince another party, the *verifier*, of the validity of a certain statement, without revealing anything more than the validity itself. The concept was first introduced in a paper by Goldwasser, Micali and Rackoff [13]. Here, authors define zero-knowledge proofs as proofs that reveal no additional knowledge other than the correctness of the statement in question. More precisely, zero knowledge proofs must satisfy three properties:

- Completeness: a honest verifier, i.e. properly following the protocol, will always accept the proof generated by an honest prover if the statement is indeed true.
- Soundness: no dishonest prover can ever convince a verifier that a false statement is true.
- Zero-knowledge: there is no action which can be performed by a dishonest verifier to extract any knowledge from a prover beyond the validity of the statement.

Zero-knowledge proofs of knowledge are a specialization of zero-knowledge proofs. Here, the prover aims at proving its knowledge of a certain secret value belonging to a statement, the validity of which is also proved. This secret value is called witness, and the verifier is convinced of its knowledge without learning anything about the value itself. For example, a prover P could prove the knowledge of a secret witness x_{secret} satisfying the statement $y = H(x)$ when $x = x_{secret}$ and $y = y_{public}$. In other words, P can produce a zero-knowledge proof of knowledge to convince a verifier V about the knowledge of a secret value x_{secret}, which is the hash pre-image of a non-secret value y_{public}. Traditionally, zero-knowledge proof of knowledge protocols require the two parties to have an interactive communication composed of several successive steps.

Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs) are a novel zero-knowledge proof of knowledge system [14, 15, 16, 17] introducing several optimizations. Among them, the fact that they are non-interactive, allowing the prover to generate proofs asynchronously with respect to the verifier. In other words, given a certain statement, a proof of knowledge for a certain witness is generated once and for all, and can be verified at any later moment by the verifier. Furthermore, they are succinct, meaning that (i) the verification process is not computationally expensive and (ii) the proofs are limited in size. The zk-SNARKs system is composed of three algorithms, *Setup*, *Prove* and *Verify*, informally defined as follows:

1. The *Setup* is performed by a trusted third party who knows the structure of the statement S that needs to be proved (meaning the size of its variables and the algorithms in use) but not the assigned values. In other words, the trusted third party does not need to know a witness for the statement, nor its public (non-secret) values. Using that structure as input, the proving key pk and the verifying key vk are generated by the trusted party. Referring to the example mentioned before, S could be defined as $y = H(x)$, and to generate the keys the third party would have to know the size of y and x, e.g. in terms of bytes, as well as the hashing algorithm H in use, e.g. the SHA256 algorithm. The proving key must then be used by any prover who wants to generate a proof for the statement S to a verifier who is employing the corresponding verifying key. It is worth noting that the trust placed into the third party is crucial. Indeed, having performed the *Setup*, this actor has the potential to generate fake proofs that would be accepted as valid by a verifier.
2. Before proceeding, P selects valid values for the variables in S. Continuing the example, P selects x_{secret} and y_{public} such that $y_{public} = H(x_{secret})$. The *Prove* algorithm takes as input the assigned values, along with the key pk, and generates the proof π.

3. The *Verify* algorithm takes π and *vk* as inputs, along with the non-secret values, i.e. y_{public} in the example. If the proof is valid, the output of *Verify* is successful, and V is convinced about two facts: (i) the statement with assigned values, x_{secret} and y_{public} in the example, is indeed satisfied; (ii) P knows the witness, i.e. the value x_{secret}. It is important to note how the secret value (y_{public}) remains unknown to V. Due to the zero-knowledge property, V is not able to extract this information in any way from P.

Chapter 3

IoT Software Updates

The goal of this Chapter is threefold. It begins illustrating, in Section 3.1, the importance of software updates for the Internet of Things, and the challenges associated with this non-trivial task. Among those challenges, we later focus on the main motivation for this thesis work: the issue of scalability in an ever-growing IoT market. In Section 3.2, several related research studies are presented, proposing solutions for delivering IoT updates. Finally, Section 3.3 is entirely dedicated to one of those related works, as it constitutes the central inspiration for our work.

3.1 Challenges for IoT Software Updates

As we argued in Chapter 1, the number of IoT devices deployed around the world is growing at incredible speed. These devices often present serious vulnerabilities that can be exploited by attackers, with significant damage. A situation which represents a threat not only for the users of the devices themselves (e.g. in terms of privacy) but also for other entities participating in the same network (e.g. DDoS attacks targeting servers on the Internet and exploiting IoT objects as zombie-agents). For this reasons, it becomes fundamentally important to regularly deliver software updates to these devices to patch their vulnerabilities, to protect their users and the network as a whole. Several studies in the literature have identified this problem as one of the main challenges faced by the Internet of Things at present times, but most importantly in the future when the number of devices will increase overwhelmingly.

A paper by Lin and Bergmann [2] investigates the challenges faced by IoT Smart Home environments in terms of privacy and security. The authors identify the issue of software updates as one of the two most critical problem for the future of those systems. They argue that while desktop operating systems and smartphones regularly receive software patches, things are different for IoT devices. Updating personal computers and general purpose mobile devices is feasible because the number of deployed devices is acceptable, still in the millions. On the other hand, IoT objects amount to several billions, and the number is growing rapidly. Furthermore, they are not homogeneous devices. For this reasons, they do not receive the same treatment, and regular patches provided by the manufacturer are not common. Additionally, manufacturers often fail to provide Smart Homes with adequate technical support. Consequently, the authors advocate for a mechanism capable of automatically deploying software and firmware updates to IoT devices. This mechanism, they argue, should require minimum user intervention, and could be easily undertaken by the home gateway.

The work of Kimani et al. [3] focuses on IoT devices that are employed as components for smart grid networks, i.e. smart electricity grids where IoT sensors and actuators can be controlled through the internet to create an intelligent eco-system, capable of providing optimized (and greener) services to users. The authors recognize the huge advantages of such intelligent systems, but they also identify the security issue as one of the most critical challenges, especially considering the specific consequences in this context. Indeed, a successful attack on a smart grid would mean significant economic loss. But most importantly, it could result in electricity shortages which

could, in turn, produce catastrophic effects for environments such as hospitals or care homes, as well as many others. The paper goes on noting that one of the main obstacles for the security of these IoT devices is the lack of a possibility to update their software once they are deployed. Currently, most manufacturers do not plan for future upgrades, even though security patches would constitute a very effective instrument to mitigate potential threats. For all this reasons, the authors underline the need for innovative approaches capable of deploying software patches effectively and sustainably.

An article authored by Ray et al. [18] focuses entirely on IoT hardware patching and arrives to similar conclusions: IoT objects are becoming so pervasive that their security is undoubtedly critical, especially for certain infrastructures such as hospitals, power grids or water supplies. Even though the article is advocating for hardware updates, it draws many parallels with software patches and offers meaningful insights that are applicable to both fields. For example, it highlights how IoT devices cannot benefit from the short life-cycle that characterizes other products. In fact, traditional computing systems such as desktop devices and smartphones are frequently replaced with new models, thus avoiding the need to fix vulnerabilities presented by old versions. Instead, IoT devices are often expected to last for decades, making it impossible to exploit device replacement as an easy alternative to fixing old vulnerable models.

Authors of [4] make similar observations in an article dedicated to the future of software updates for the Internet of Things. IoT devices, they note, are often deployed with a fixed firmware, without any plan to upgrade it in the future. This trend is expected to change in the coming years, when hardware-independent IoT software will naturally and increasingly spread. Furthermore, the authors underline the discrepancy between (i) the tendency of vendors to provide updates just until the end of the warranty period and (ii) the long life-expectancy of many IoT objects such as smart washing machines. Hence the need for protocols and platforms allowing manufacturers to easily provide software updates without incurring in the high costs that usually hinder this deployment for longer periods.

Having recognized the importance of keeping IoT devices updated in an efficient and secure way, the task itself remains a very difficult one. Indeed, the challenges are numerous and diverse.

The work of Hernandez-Ramos et al. [5] aims exactly at enumerating and analyzing those challenges, as well as describing some potential solutions. The authors acknowledge the same premise that was presented so far: IoT devices are becoming more and more ubiquitous and therefore very attractive targets for attackers, and patching them is an essential part of the necessary protection. Among the many heterogeneous issues identified by the authors, we will present the ones we focused on throughout this thesis. Those are also among the set of challenges we addressed in our proposed protocol.

One of the most important issues is integrity. It is fundamental that any update installed by an IoT device has not been tampered with, otherwise the patching process would create more threats than it mitigates, as it could lead to malicious code being executed.

Another challenge is identified in the problem of trust. That is, IoT devices must be able to trust software providers, to avoid illegitimate software being released from unauthorized entities. In particular, this can be achieved through the use of proper cryptographic tools and key management algorithms.

Furthermore, the authors point out how the restricted resources that usually characterize IoT objects should also be considered. Cryptographic primitives and protocols should be efficient and lightweight, requiring the minimum amount of computing power for IoT entities and using small overheads in their messages to avoid overloading the network.

Finally, the paper underlines an issue that we embraced as one of the main motivations for our work. That is, the issue of scalability and the consequent need to transition from current centralized approaches to decentralized solutions. The authors note how, nowadays, the majority of manufacturers deliver updates using traditional client-server architectures. However, given the rate at which the number of IoT devices is growing at globally, this approach is not sustainable for the future. IoT manufacturers would be required to maintain ever-growing data centers to distribute software patches to billions of devices, incurring in incredible costs. Several studies in the literature recognize this problem as a crucial one, and propose different solutions. In section 3.2, we will describe and analyze those proposals.

3.2 Related Work

Boudguiga et al. [6] propose the use of a blockchain infrastructure to deliver updates to IoT devices belonging to different manufacturers. The authors do not provide one single solution, but many alternative approaches that can be employed to increase the availability and the integrity of updates. In practice, they suggest two main strategies.

The first is for the manufacturer to sign a certain update that has to be delivered, and upload it directly on the blockchain. This approach would provide a very important advantage: persistency. That is, to ensure, thanks to the built-in properties of the blockchain, that once an update file is uploaded it will remain there forever and unaltered. What is more, the blockchain would also provide more resistance to common availability threats, such as Denial of Service (DoS) attacks, compared to traditional client-server infrastructures. Finally, it could solve the scalability problem, given the decentralized nature of blockchain technologies and protocols. However, the paper itself recognizes the limits of this strategy: uploading the whole update file would easily become unfeasible due to the blockchain size, which would have to be downloaded by all participants in its entirety. This is especially true in the context of IoT, where the number of devices is so significant and their nature is also very heterogeneous. What is more, IoT devices are often limited in terms of hardware and software capabilities, including storage.

For this reasons, they also propose a second strategy, where the blockchain is limited to ensuring what they call update innocuousness. That is, the ability to provide all the information needed to verify the integrity of an update without storing the update file itself, which is the most inconvenient element in terms of size. Additionally, as part of this second strategy, the authors propose to exploit peer-to-peer (P2P) technologies to distribute the actual update file, in parallel to the blockchain framework. In this context, IoT devices would behave as nodes in the P2P network and share the file with other peers that need the same update.

We argue that the first strategy, as the authors themselves acknowledge, can too easily become inapplicable. Uploading the actual patches to the blockchain would increase its size at an unsustainable rate. On the other hand, we consider the second strategy to be very effective, as it solves the scalability issue exploiting P2P file sharing.

However, this approach presents two main issues. First of all, it suffers from the same problem that usually affects P2P file sharing protocols such as BitTorrent. That is, the tendency to only download from other peers without ever uploading, leading to low availability for files. Hence, the need for a mechanism able to stimulate users to actively participate in the network, committing their resources and sharing files. In traditional P2P file sharing settings such as BitTorrent, this mechanism is known as chocking. That is, a punishment for users that do not behave correctly: when other peers notice their misbehavior, they stop uploading data to them. But this solution is not applicable in this context where IoT devices are interested in downloading just one particular update file in a given time window. Indeed, once a device has obtained the update file, it has no interest in uploading it to others. And the disincentive of bandwidth-choking would not work because this device has no interest in downloading anything else until the next update is released.

Secondly, we argue that the framework proposed by Boudguiga et al. is not taking the IoT resource constraints into consideration. In their solution, IoT devices are expected to actively participate as full nodes in both the blockchain network and the P2P network. While this task is arguably trivial for traditional devices such as desktops or even smartphones, the same is not true for IoT objects, which are often battery-powered and need to preserve as much energy as possible by limiting their computational efforts.

Another work by He et al. [7] also tries to exploit the benefits of blockchain technologies, focusing on Over-The-Air (OTA) firmware updates for IoT. They design a mechanism that allows IoT devices to validate new firmware releases securely. Manufacturers are required to deploy a new transaction to the blockchain whenever a new update is released, containing important information that can be consulted by an interested IoT device later. The next step is for the manufacturer to send the update itself to the IoT device, along with the transaction ID that will indicate where to look in the blockchain. In this way, the IoT device can check the integrity and the authenticity of the update before installing it.

We believe this approach to be very effective in terms of validation. Blockchain does indeed provide the perfect characteristics for this purpose, successfully eliminating any possibility of tampering with the firmware update file. However, the solution proposed by the paper does not address the scalability problem we identified earlier. Despite having recognized the importance of not uploading the update file directly to the blockchain, the authors do not propose an alternative approach, implicitly suggesting the use of traditional centralized mechanisms to deliver the actual update file that will be later validated by means of the proposed algorithms.

The next solution proposed by Lee [8] presents several improvements, compared to the previous studies. The paper identifies the scalability issue, and the consequent need for new decentralized update delivery systems, as the main motivation for its contribution. The proposal consists in a new decentralized framework capable of incentivizing untrusted third parties to participate in the delivery of updates, with the security requirements being enforced through blockchain and smart contracts. This solution has three main participants. The *provider* corresponds to the manufacturer, the entity whose objective is to deliver a certain software update to a specific set of IoT targets called *recipients*. And then, an intermediary third-party entity is introduced, called *transporter*. Transporters take care of (i) obtaining the update files from the provider (or providers) and (ii) delivering them to the recipients. The basic idea is that a transporter will get a reward, in the form of cryptocurrency, for every update package delivered to each recipient. The fairness of this process is enforced through the blockchain (and the smart contracts deployed on it) and through other traditional cryptographic mechanisms.

An instance of the protocol can be divided in four steps:

1. First of all, a provider (wanting to deliver an update to a set of recipients) encrypts the update file U with a key s that is known only by the provider itself and the recipients. Then, another round of encryption is applied to the update, this time with a different key k_i, obtaining U_k. Afterwards, the provider prepares an update package containing many elements, including the key k_i and U_k. It is important to note that the provider creates a different package for each recipient, i.e. for each different target IoT device. In other words, if the provider wants to deliver the update to a number n of devices, it will have to create n different packages, each with a different key k_i corresponding to a certain recipient device D_i. The package also contains some metadata, such as the hash of the key $H(k_i)$. Finally, it includes a keyed hash that will allow the recipient to verify the integrity of the elements. A keyed hash is generated by hashing a concatenation of (i) various elements that need to be validated and (ii) a specific secret key known only to the other party who will perform the validation. In this case the secret key is s and the elements are the ones that will be sent to the recipient, including U_k and $H(k_i)$.

2. At this point the transporters receive the packages. Each package, different from the others because of a different key k_i, can be used to deliver the update to a specific device D_i. Once a device has been found, the transporter can send the package after having (i) removed the key k_i and (ii) added its own blockchain address. The latter will be used to set up the payment of the cryptocurrency reward. This modified package will be then validated by the recipient by means of the keyed hash mentioned before.

3. If the modified package is successfully validated, it means that the hash of the key $H(k_i)$ and encrypted file U_k are indeed authorized by the provider by means of the keyed hash. In this case, the recipient can generate the receipt, which consists in a smart contract deployed to the blockchain. The smart contract code will perform two main actions: (i) receive the decryption key k_i from the transporter and check its validity, mainly checking if its hash value corresponds to the hash value that was received in the modified package by the recipient; (ii) send the reward to the transporter and publish the key k_i, so the recipient can decrypt U_k. The reward is sent (and the key is revealed) only if the validation is successful.

4. Finally, the transporter can send the key k_i to the smart contract. If the key is valid, the contract code will securely and atomically (i) send the cryptocurrency reward to the address that was specified by the recipient at contract creation and (ii) publish the key for anyone to read. The recipient will then be able to decrypt the update file.

The solution proposed by this paper constitutes a very fine proposal for a decentralized update delivery system, which has many aspects in common with our own work. The manufacturer can do away with expensive client-server architectures and can delegate this task to (untrusted) third parties, incentivizing them through micropayments in the form of cryptocurrency. What is more, the fairness of the protocol for both the recipients of the service (IoT devices) and the transporters is automatically enforced by means of the blockchain and the smart contracts deployed on it. And the same goes for the security of the framework: assuming that the provider is behaving honestly, there is no way for the other entities to perpetuate malicious attacks. However, we identified some flaws in this design.

First of all, IoT devices are required to perform many different actions, which are often computationally expensive. This is especially true if you consider the constraints on IoT resources, both in terms of hardware and energy consumption. The most significant example is the need to function as full blockchain nodes. That is, the need to be able to monitor the blockchain, commit transactions and deploy smart contracts. The latter also includes attaching a certain amount of cryptocurrency as a fund for the contract itself, which means that IoT objects are responsible for managing a cryptocurrency wallet, adding even more complexity. For this reasons, we argue that this framework could not be suitable for many IoT devices.

Secondly, the protocol is designed so that every IoT device receiving an update file will have to deploy a new smart contract. This entails a computational burden, as we already mentioned, but also a cost in terms of cryptocurrency. Each new deployment will require a new fee, and given the high number of devices, the cost could easily become overwhelming.

Finally, the paper presents a major issue related to one of its main objectives. That is, scalability. Even though the proposed framework successfully manages to delegate the delivery of updates to third-party transporters in a fully decentralized way, there is a structural problem preventing the system from scaling in practice. The protocol is constructed in a way that requires the manufacturer to produce a different package for each recipient. In other words, the number of packages that need to be prepared and sent out is equivalent to the total amount of target IoT devices. The number of transporters could be much lower. It could even amount to just one transporter, delivering the update to all targets. However, the manufacturer would still have to send n different updates packages for n devices. We argue that, in this way, the purpose of avoiding a client-server architecture for distributing the update is defeated. Indeed, sending the update n times to a set of intermediate third-parties would be even more costly than the traditional centralized approach.

The next related research is also the last of this kind that we present in this thesis. Compared to the previous proposal, it brings many novelties and optimizations, and comes a step closer to the main objective we set for ourselves: designing an update delivery system capable of scaling with the rising number of IoT devices being deployed around the world. Furthermore, this next paper constitutes the main inspiration for our own design. For all this reasons, we will illustrate it separately in the next section.

3.3 Our Case Study

This section is entirely dedicated to illustrating the work of Leiba et al. [9]. Their design of an Incentivized Delivery Network of IoT Software Updates Based on Trustless Proof-of-Distribution can be regarded as the basis for our contribution.

In few words, their proposal consists in a highly decentralized framework that allows any manufacturer to delegate the burden of delivering IoT software updates to self-interested third-parties. The honest behavior of these parties, called distributors, is enforced through (i) the use of smart contracts deployed on a blockchain and (ii) a powerful zero-knowledge tool called zk-SNARKs. Both tools were discussed in Chapter 2.

We will first outline the entities participating in this framework, in Section 3.3.1. Afterwards, in Section 3.3.2, we will illustrate the steps of the protocol. In Section 3.3.3 we will finally underline a set of issues we identified in this work, as well as the corresponding modifications and improvements we are proposing.

3.3.1 Framework Entities

First of all, the framework requires the existence of two important underlying infrastructures: the blockchain network and a peer-to-peer (P2P) file sharing network. The blockchain network must support smart contracts and digital currency. And it must be permissionless, meaning that any interested party can read and post messages. The authors believe Ethereum to be the best real-world example of a blockchain network fitting all those requirements. From now on, we will implicitly refer to the Ethereum implementation when discussing blockchain-related mechanisms. As of the P2P network, it must be accessible to anyone, allowing files to be consumed and served by means of a peer discovery scheme such as a distributed hash table (DHT).

Another important entity is the vendor, i.e. the manufacturer who initiates any instance of the protocol by releasing a new update. It must be able to participate both in the blockchain network and in the P2P network. In particular, it must be a full blockchain node, capable of maintaining a wallet and initiating transactions. Furthermore, any manufacturer m must possess an asymmetric key pair, i.e. a private key prv^m and a public key pub^m derived from prv^m.

Each vendor m is the manufacturer of a specific set of IoT objects and manages their updates. For this purpose, the vendor stores a list of public keys $pub^{o_{m,i}}$ each belonging to a device $o_{m,i}$. In addition to its public key, each IoT object is also in possession of the corresponding private key $prv^{o_{m,i}}$, as well as the private key of the manufacturer. IoT objects have less requirements, compared to vendors, when it comes to the participation in the blockchain network. That is, they are not expected to store an entire copy of the blockchain data structure, to avoid incompatibility with certain resource-constrained IoT devices. The authors argue that this lightweight approach can be achieved in various ways, such as relying on a trusted blockchain node or employing the Ethereum light client modality that is currently under development. The situation is similar for IoT devices participating in the P2P network: they are only required to be able to download files from other peers, while they are not required to upload anything.

Finally, the remaining entity involved in this framework is the distributor. Distributors are third-party untrusted agents that are interested in delivering the update file to the IoT objects in exchange for a cryptocurrency reward. They are required to be full participants in both the blockchain network and the P2P network. Literally anyone can act as a distributor as long as they meet those two requirements. Each distributor d_j also needs to posses a private key prv^{d_j} and the corresponding public key pub^{d_j}.

3.3.2 Protocol Steps

We illustrate the steps needed to successfully terminate an instance of the protocol in which a single manufacturer m is intentioned to release an update file U destined to a set of IoT objects $o_{m,1}, o_{m,2}, \ldots, o_{m,n}$.

Update Release

Manufacturer m is about to release an update file U and performs the following steps:

1. Computes the hash of the update $U_h := H(U)$ which will serve as an ID for the file.

2. Generates the zk-SNARKs keys: the proving key pk and the verifying key vk.

3. Prepares the update package P containing the update itself and other useful information.

   Formally, the package is defined as $P := \{U, vk, pk, sig_m\}$ where sig_m is a signature made by the manufacturer m on the concatenation of U and vk, using the private key prv^m to generate the signature itself. To be precise, the signature is defined as $sig_m = Sign_{prv^m}(U_h||vk)$.

4. Computes the hash of the package $P_h := H(U)$.

5. Decides the duration Δ of the time-window in which the update can be delivered.

   The purpose of this value will be clear later. In short, it defines a time-window, the purpose of which is twofold: (i) a distributor can claim a reward only within this time and (ii) the manufacturer is not allowed to withdraw the funds committed in the smart contract before this window expires. This is used to enforce fair conditions for both distributors and vendors. For the latter, it allows to set a deadline after which the funds committed to the smart contract can be withdrawn, e.g. in case a subset of IoT devices is unreachable and the corresponding rewards cannot be claimed by anyone. For the distributors, it avoids a situation in which they deliver the update but cannot obtain the corresponding reward because the vendor withdrawn the funds from the contract.

6. Finally, the manufacturer deploys a smart contract to the blockchain.

   This is a smart contract corresponding to this specific update release for the selected list of IoT devices. In other words, the manufacturer would need to deploy a new smart contract for any new update, or for the same update but with a different set of devices.

   The vendor m needs to provide several elements to deploy a well-formed smart contract. First of all, it needs to send an amount f of cryptocurrency as a deposit. This deposit will constitute the smart contract funds and will be used to pay the single rewards to distributors. As mentioned before, this fund cannot be withdrawn by m until an amount Δ of time has passed. This amount is also provided to the contract at creation time. Other elements needed for its deployment are: the update hash U_h, the package hash P_h and the list of public keys $pub^{o_{m,1}}, pub^{o_{m,2}}, \ldots, pub^{o_{m,n}}$ belonging to the target IoT objects manufactured by m. All these elements are public, readable by anyone who can access the blockchain.

   When creating a new smart contract, the manufacturer also needs to provide its code, i.e. the logic of the smart contract itself, which will be securely executed by the blockchain network through the distributed mechanism described in Section 2.1. The authors define this logic by means of a pseudo-code fragment. In simple terms, it results in the following algorithm. For each public key $pub^{o_{m,i}}$ in the provided list, if any distributor with public key pub^{d_j} is able to present a proof-of-delivery $sig_{o_{m,i}}$ before the expiration of Δ, then a payment will be sent to that public key pub^{d_j}, using the deposit f as a source of cryptocurrency to fund the payment. The nature of the proof-of-delivery $sig_{o_{m,i}}$ will be explained later.

Initial Seeding

At this point, distributors become aware of the new release, as they are regularly monitoring the blockchain. The manufacturer enters a temporary phase in which the update package P is sent, via the P2P network, to any distributor who requests it. In particular, the request is performed using the hash value P_h readable from the smart contract. This phase is meant to last a limited amount of time to avoid high costs for the vendor in terms of bandwidth. After its end, a certain amount of distributors will have obtained the update package P. It is worth nothing that from this moment on, there is no way for any additional distributor to participate in the distribution of this particular update release taking place in this instance of the protocol.

Upon obtaining the package P, any distributor performs two checks. First, it computes the hash of the package P and compares it to the hash value P_h readable from the blockchain to check its integrity. Secondly, it checks the validity of the signature contained in the package using the public key of the manufacturer.

Update Delivery

After the initial seeding phase has come to an end, distributors are ready to deliver the update. They announce its possession on the P2P network by enlisting its hash value U_h. At the same time, IoT objects, who are also regularly monitoring the blockchain, become aware of the new release. They turn to the P2P network and request the update using its hash value U_h retrievable from the smart contract.

Let us analyze the steps performed in this phase by a distributor d who has received a request for the update from an IoT object o_m. This object belongs to the set of many IoT devices manufactured by m and its public key is included in the list stored by the smart contract. At the end of the following steps, the distributor will have obtained the proof-of-distribution required to claim the cryptocurrency reward. On the other hand, the IoT device will have obtained the encrypted version of the update. The key to decrypt it will be sent by the distributor to the smart contract to unlock the reward payment, and the smart contract will publish it for anyone to read. As we will see, the fairness of the protocol is enforced through zk-SNARKs and the contract code.

1. Distributor d sends an identification challenge c to object o_m.
2. Device o_m performs a signature on c and sends the tuple $(idsig_{o_m}, pub^{o_m})$.
   Formally, the signature is defined as $idsig_{o_m} := Sign_{prv^{o_m}}(c)$
3. Distributor d performs a series of sub-steps:
   (a) Verifies that pub^{o_m} belongs to the list in the smart contract.
   (b) Verifies the signature $idsig_{o_m}$.
   (c) Securely generates a random key t.
   (d) Computes the value $r = H(pub^d||t)$.
   (e) Computes the value $s = H(r)$.
   (f) Obtains $U_e = Enc(U, r)$ by applying symmetric encryption to U with r as key.
   (g) Generates the zk-SNARKs proof π for the following statement S using pk:

   $$s = H(r) \wedge U_h = H(U) \wedge U_e = Enc(U, r)$$

   As explained in Section 2.3, the generated proof π works as follows. It is produced by a prover, in this case d, with the objective of proving the validity of a statement, in this case S, without revealing all the elements constituting the statement itself. The elements that are revealed are referred to as public values, the others are called secret values. In this context, the secret values are U and r, while the public values are U_h, U_e and s. To generate the proof, a specific zk-SNARKs function needs to be used. This proving function takes as input both the secret values and the public values, and the proving key pk. The latter must be produced by a trusted third-party. In this scenario, pk was produced by the manufacturer along with the verifying key vk, both included in the package P. The verifying key vk is later used by the verifier as input for the verifying function, along with the public values and the proof π.
   (h) Finally sends a message to o_m containing: the public values U_h, U_e and s; the zk-SNARKs proof π; the signature sig_m that was included in the package P.
4. Object o_m receives the message from d and performs the following sub-steps:
   (a) Verifies the signature sig_m to authenticate the values U_h and vk.
   (b) Verifies the zk-SNARKs proof π of the statement S.
   If the proof is valid, o_m will be convinced about the validity of the statement S. In other words, there will be mathematical assurance about two important facts. (1) The encrypted update U_e received from d was effectively obtained encrypting an update file U with hash value U_h using a key r. It is important to remember that the hash value U_h was authenticated by the vendor by means of the signature sig_m, and that the key r remains unknown to o_m for the time being. (2) The received hash value s was indeed obtained computing the hash of the key r. As we already said, the value r is proven to be the key used to encrypt U and obtain U_e.
   (c) Sends the proof-of-delivery $deliverysig_{o_m}$ to d
   This proof-of-delivery is simply obtained as a signature on the concatenation of two values, U_h and s, i.e. $deliverysig_{o_m} = Sign_{prv^{o_m}}(U_h||s)$. On the other hand, the importance of this signature for the protocol is quite remarkable. It can be seen as a formal commitment made by the IoT device, a digital token capable of unlocking the cryptocurrency reward for d. This will be clear in the next steps, when the distributor will send the proof-of-delivery to the smart contract.

Reward Claim and Key Publication

Let us continue to follow the actions of the IoT object o_m and the distributor d. The latter is now ready to claim the cryptocurrency reward as a payment for its delivery efforts. When the following steps are successfully executed, two events are triggered atomically: (i) a payment is issued by the smart contract to the distributor and (ii) the key is published on the blockchain, allowing the IoT device to finally decrypt the update file which is already in its possession.

1. Distributor d verifies the signature $deliverysig_{o_m}$
2. Now d can post a transaction to the smart contract on the blockchain, attaching various elements: its own public key pub^d; the public key pub^{o_m} of the IoT object; the values t, r and s; the proof-of-delivery $deliverysig_{o_m}$.

   The smart contract code takes care of validating this transaction in a secure and fair manner, checking that all those values were produced correctly. The transaction is successful only if the following conditions are satisfied: (i) the time-window is not expired; (ii) the public key of the IoT object belongs to the list provided by the vendor at creation time; (iii) the update has not yet been delivered to this specific object; (iv) the values t, r and s have valid mathematical relationships as in the statement S; (v) the value $deliverysig_{o_m}$ sent by d is indeed a valid signature.

   If the transaction is considered valid by the smart contract, a payment will be automatically issued to d, and the key r will be published to the blockchain.
3. IoT object o_m retrieves the key r from the blockchain and decrypts $U = Dec(U_e, r)$.

3.3.3 Discussion

Compared to the last related work discussed in Section 3.2 [8], this paper presents several significant improvements. We outline them in the following.

First of all, there are less requirements for IoT objects to fully participate in the protocol. IoT devices are not supposed to maintain a cryptocurrency wallet, for example, since the deposit made by the manufacturer is used by the smart contract to issue reward-payments instead. Also, they are not required to store the entire blockchain data structure locally: it is enough for them to rely on a trusted gateway or to act as light nodes. Another major optimization is the number of smart contracts generated per instance of the protocol. Instead of producing as many contracts as the number of target IoT devices for each update release, only one contract is deployed for any single release, independently of the number of devices.

Furthermore, there is a significant improvement in terms of scalability. The manufacturer is not required to distribute as many update packages as the number of IoT objects. Instead, there is only one package per update release. This means that, hypothetically, the vendor could send just one package to only one distributor. And that distributor could bring the update file to the entire set of target IoT devices. In this case, the vendor's bandwidth efforts would be incredibly minimized. Despite being an extreme and improbable scenario, it is indeed theoretically possible given the design of the protocol. The same cannot be said for the other framework.

However, we identified a series of issues in this design, which will be illustrated in the following. As a consequence to these findings, we conceived several important modifications, which constitute the contributions of this thesis. These enhancements will be mentioned here and then discussed in depth in Chapter 4. They concern different aspects of the framework, including its scalability, its security and its efficiency.

First of all, during the initial seeding phase, only a finite number of distributors will obtain the update file from the manufacturer. That is because any given distributor has no interest in sharing the file with others, since it would lessen its possibility to obtain a reward. We argue that this limitation does not allow the framework to adequately fulfill the goal of improved scalability compared to current centralized approaches. Indeed, the number of distributors can only grow linearly with the size of the time-window in which the manufacturer seeds the file. To solve this

issue, we introduce the possibility for distributors to share the update file with new distributors in exchange for a direct payment in the form of cryptocurrency. We argue that this improvement will effectively boost scalability, as it allows the set of distributors to grow exponentially.

Another issue is that, even if the requirements for IoT devices are indeed quite reduced compared to other works, they are still significant for such resource-constrained entities. For example, in terms of their involvement with the blockchain network. Even if they can avoid storing the entire blockchain data structure, they are still expected to monitor and access it. This results in the need to implement a whole application layer to understand and interact with the blockchain, which is quite a burden in terms of software complexity. A similar reasoning is applicable to the P2P network. Despite the fact that IoT objects are not required to share any file to other peers, significantly reducing their requirements compared to traditional peer-to-peer file sharing environments, they are nevertheless required to interact with distributors through the P2P network. This adds an unnecessary layer of complexity to them, especially compared to traditional client-server architectures in which a device simply establishes a direct communication link with its vendor to download an update. Lastly, IoT objects are also required to verify zk-SNARKs proofs. This operation is much less complex than generating such proofs, but still very demanding for this type of devices. To overcome this problem, we introduce a new participant to the protocol: the hub. That is, a gateway device managing a heterogeneous set of IoT devices all connected to the same local network. In general, many studies show the importance of the hub in the IoT context, including from a security perspective. Cirani et al. [19] propose the use of a hub as a way of managing heterogeneous IoT devices. Another example is RES-Hub, a solution proposed by Doan et al. [20] in which the goal of the hub is to bring resilience to the IoT ecosystem by providing functionalities when the cloud is unavailable. Authors in [21] propose PLAR, a PLuggable And Reprogrammable software architecture aiming at securing IoT devices, in which a hub plays a key role for achieving this goal. Finally, the solution proposed by Simpson et al. [22] employs a hub as a central security manager. Furthermore, many current real-world IoT ecosystems employ this entity: devices are usually not directly connected to the Internet, but instead managed by an intermediate hub or gateway which is a more capable device in terms of computational power, storage and bandwidth [23]. Examples are the Apple HomeKit[1] or the Samsung SmartThings hub[2]. In our protocol, hubs are responsible for performing the majority of the steps necessary to exchange the update file for a proof-of-delivery. On the other hand, we also allow the case in which an IoT device is capable enough to perform these steps by itself without the aid of a hub. Additionally, to incentivize the participation of this entity in our framework we introduce another type of reward meant only for hubs.

Moreover, we identified a potential vulnerability in the framework. That is, the lack of a countermeasure for malicious distributors pretending to possess the update file to waste the resources of an IoT device. Despite the incentive of the cryptocurrency reward, which should encourage distributors to behave honestly in most cases, we argue that this attack could still be performed to prevent specific IoT targets from obtaining the update. Therefore, we designed a protection mechanism: a simple database deployed on the blockchain, recording which distributors are behaving honestly. This is implemented with integer values associated with the public keys belonging to distributors. These values are recorded in the blockchain automatically by the smart contract, making the database secure against any tampering. In this way, an hub or an IoT device can consider these scores to choose the most trustworthy among a set of available distributors.

Finally, while performing the formal analysis of our protocol, we discovered an important vulnerability that is also applicable to this design. That is, the challenge c sent by the distributor to the IoT device for identification purposes (see Section 3.3.2). This challenge can be constructed in a way such that the IoT device will directly produce the proof-of-delivery needed for the distributor to get the reward. In other words, by simply choosing the right value for c, a distributor can obtain the cryptocurrency payment without actually delivering the update, effectively breaking the security of the protocol. For more information, see Section 5.2.3 where we discuss the corresponding segment of the formal analysis of our protocol.

[1] https://support.apple.com/en-us/HT207057

[2] https://www.smartthings.com/gb/products/smartthings-hub

Chapter 4

Proposed Protocol

In this thesis we propose *CrowdPatching*, a secure protocol allowing manufacturers to do away with expensive centralized infrastructures in order to deliver software updates to their IoT devices. Instead, *CrowdPatching* consists in a fully decentralized system in which the distribution of IoT updates is fully decentralized, being delegated to a scalable set of self-interested agents. These actors, called distributors, are rewarded through cryptocurrency micro-payments for each single IoT device they are able to deliver an update to. The cryptocurrency payment-system, although funded directly by the manufacturer, is managed automatically through the use of blockchain smart contracts to ensure fairness for all the parties involved.

4.1 Participating Entities

4.1.1 Blockchain Platform

The protocol requires the presence of an underlying blockchain platform with specific features. First of all, it must be permissionless. As explained in Section 2.1, this refers to the possibility for anyone to access the blockchain freely. More precisely, there are three main modalities to access the blockchain. The first is to simply read the data on the public ledger, without interacting with it in any other way. This can be done by any node connected to the blockchain network, without any other requirement. A second access mode is to act as a passive node, posting transactions to the blockchain network without taking care of mining new blocks. The only requirement to perform this type of action is the possession of a public-private key pair. The identity of a node is represented by its public key, and any transaction posted by this node must be accompanied by a signature on the transaction itself, generated with its private key. Finally, nodes can participate as active nodes, called miners, gathering transactions from other peers and validating them. In the proposed protocol, participating entities are not expected to act as miners. Instead, they only read from the blockchain or, at most, post transactions to the network.

The blockchain structure must also have native support for cryptocurrency. In fact, after Bitcoin was introduced, many alternative blockchain implementations were proposed that did not provide support for digital currency exchanges. However, this feature is extremely important for this protocol. Indeed, some participant nodes are required to post transactions to the blockchain network which involve cryptocurrency payments.

Another required feature is for the blockchain to support smart contracts, as described in Section 2.2. That is, it must support the deployment of computer code to the blockchain, which is executed automatically in a distributed fashion by the network and can be triggered by specific transactions or events. What is more, smart contracts need to have the ability to generate new smart contracts. In other words, it must be possible for a user in the blockchain network to structure the code of a smart contract in such a way that, when certain conditions are met, this will automatically deploy new contracts to the same blockchain network, without any aid from the original creator, i.e. the user who programmed the first contract.

A real-world platform implementing all these features is the Ethereum blockchain. Throughout the rest of this chapter, we will implicitly refer to the Ethereum implementation when discussing blockchain-related mechanisms of the proposed protocol.

4.1.2 Peer-to-peer file sharing

A subset of the participating entities of our protocol make use of a peer-to-peer network [24] to share files, i.e. a peer-to-peer file sharing network. More precisely, this network is decentralized and must be accessible to anyone. What is more, it must provide a peer-discovery scheme based on a distributed hash table (DHT). That is, a distributed data structure providing a lookup service in the form of a table, with key-value pairs as entries. In this system, any user can lookup the address of other peers holding of a certain file f by means of its hash value $f_h = H(f)$. In order to be found, these peers must announce their possession of the file to the network, which results in the addition of the mentioned entry in the DHT. This type of network, with the indicated peer-discovery scheme, provides a number of benefits. These include (i) fault-tolerance, as the system is strongly decentralized and does not present a single-point-of-failure, and (ii) scalability, as it allows the participation of huge numbers of nodes simultaneously. The details of such peer-to-peer network are out of scope for this thesis. However, several real-world implementations are available, such as the BitTorrent protocol[1].

4.1.3 Manufacturers and IoT Objects

The key objective of our proposed protocol is to allow manufacturers to deliver software updates to their IoT devices. As we will explain in a later section of this chapter, any instance of the protocol begins with a manufacturer node releasing a new update, targeting a specific set of IoT objects as recipients of such update. In this regard, it is worth noting that the protocol allows an arbitrary number of different manufacturers to simultaneously release a new update independently from each other, targeting different sets of IoT devices.

Any manufacturer node is required to maintain a private-public key pair, as well as the list of public keys belonging to all IoT devices that might be targeted by an update release in the future. Furthermore, manufacturers are expected to participate both in the blockchain network and the peer-to-peer file sharing network. In the blockchain network, manufacturers need to be able to post transactions and deploy smart contracts. In the peer-to-peer network, they are required to upload data to other peers. Manufacturers must also be able to perform various types of computations. (i) They must be able to apply symmetric encryption to files with different sizes. (ii) They need to execute the *Setup* algorithm of the zk-SNARKs system to generate a proving key and a verifying key. (iii) And they are required to compute hash values and digital signatures. Among these requirements, the most demanding is the execution of the zk-SNARKs Setup algorithm, which is quite computationally expensive. However, this algorithm is executed only once per update release. In other words, it is performed only once in order to deliver the update to a large number of devices.

IoT nodes have fewer requirements. They are still expected to securely maintain a private-public key pair. Also, they need to be capable of computing digital signatures and hash values. However, they have no other requirement. In particular, they are not required to interact with the blockchain in any way. Also, they can avoid to verify zk-SNARK proofs, as well as to participate in the peer-to-peer file sharing network. These reduced requirements are crucial for the feasibility of our protocol. As we discussed in Section 3.1, IoT devices often present restricted resources in terms of both software and hardware capabilities, therefore requiring lightweight protocols with minimum computing power demand. What is more, these devices are usually battery-powered, thus requiring low computational efforts to preserve as much energy as possible. In our protocol, it is indeed possible to avoid the computational burdens listed above, thanks to the entity described in the next Section. That is, a gateway device managing various IoT devices. These gateway devices are in charge of performing the mentioned actions on behalf of IoT objects.

[1] https://www.bittorrent.org/

4.1.4 Hubs

As discussed in Section 3.3.3, the proposed *CrowdPatching* protocol includes a new entity with respect to related research works. That is, the hub, a gateway device managing a set of devices deployed in a local network. A set of devices which can potentially be heterogeneous, meaning that managed devices could belong to different manufacturers and therefore different software updates providers. As we argued, the usefulness of hubs in the IoT context is well documented by several academic studies, including for security reasons.

We assume hubs to be trusted by the corresponding IoT devices. In other words, the IoT objects managed by a hub are expected to trust the hub itself. That is because we assume these gateway devices to be manually configured by the same user who is also the owner, or supervisor, of the IoT objects. For example, let us consider a smart home environment with a few deployed IoT devices. The owner of such home, can deploy an additional gateway device, or hub, and configure it to act as a manager of the IoT devices connected to the same local network. However, manufacturers cannot share the same trust. As shown later in the detailed protocol steps, manufacturers make no security assumptions whatsoever about hubs. This is reflected in a specific feature of our protocol: a cryptocurrency incentive encouraging hubs to behave honestly. This mechanism is explained in Section 4.2.5.

The main requirements for hubs are as follows. First of all, they need to possess a private-public key pair. They also need to be able to read data from the blockchain, as well as to post transactions on such platform. Participation in the peer-to-peer file sharing network is expected as well. They need to be able to verify zk-SNARKs proofs. And finally they must be capable of decrypting a file that was obtained through symmetric encryption. Other minor requirements include signature verification and hashing.

4.1.5 Distributors

Finally, a key role in the protocol is played by the last entity: distributors. These are self-interested agents whose objective is to obtain cryptocurrency payments in exchange for delivering software updates to IoT devices. By self-interested, we mean that they are not affiliated with manufacturers in any way. Instead, their actions are purely motivated by the prospect of obtaining the cryptocurrency payments. These payments, which can be seen as rewards, are provided by the manufacturers. As we will see later, a reward is offered for each IoT device who is targeted by a new update release. Consequently, each of these payments is automatically issued to the first distributor who is able to prove to have successfully delivered the update to the corresponding device. This mechanism is made possible by smart contracts.

Distributors are further divided in two sub-categories, depending on the modality in which they obtain the update before distributing it. This is because in the *CrowdPatching* protocol distributors have two possibilities in this regard, for each new update release. First, they can download the update directly from the manufacturer, which is the most desirable option for them. Distributors belonging to this category are called *first-hand distributors*. However, for reasons that will be explained later, it is possible for a distributor to be unable to pursue this option. The alternative is to acquire the update from a first-hand distributor in exchange for a cryptocurrency payment. Distributors following this alternative are referred to as *second-hand distributors*. Once they acquire the update, second-hand distributors become effectively identical to first-hand distributors.

As far as the security assumptions are concerned, distributors of both kinds are completely untrusted by all the other participating entities. As suggested by the name of the *CrowdPatching* protocol itself, its goal is to essentially crowd-source the delivery of IoT updates. For this reasons, literally anyone is allowed to act as a distributor. Which is why they are treated as potentially malicious actors in all protocol steps. Several cryptographic tools are used to enforce their honest behavior, as well as native protections offered by the blockchain platform.

While the protocol theoretically allows anyone to act as a distributor, this is not completely accurate in practice because of the requirements they have. Indeed, they need to be able to

perform several actions, often computationally demanding. They need to participate in the peer-to-peer file sharing network, as well as in the blockchain network. In the latter, they not only need to monitor its events, but also to post transactions. They also need to maintain a private-public key pair. Most importantly, they are expected to generate zk-SNARKs proofs.

4.2 Protocol Steps

The *CrowdPatching* protocol allows an arbitrary number of manufacturers to manage the delivery of software updates to their IoT devices simultaneously and independently. In other words, the steps illustrated in this section can be performed by any manufacturer at any time, without considering the actions of others. For this reason, in the following we will consider a single manufacturer m, to set an example that is valid for any manufacturer. An overview of all steps (with few omissions) described here is presented in Figure 4.1 at the very end of this section.

4.2.1 Super Smart Contract

Before any update can be released, the manufacturer m is first required to perform a preliminary step and deploy what we call a *Super Smart Contract* (SSC). That is, a smart contract which is capable of generating new smart contracts with a specific template. The generation of these *derived* smart contracts is triggered when specific transactions are sent to the SSC. This preliminary step is performed only once for each manufacturer.

The SSC has an additional purpose: it stores an integer score associated with any distributor. More precisely, an SSC maintains a data structure in the blockchain keeping track of successful deliveries accomplished by distributors, by means of an integer value associated with their public keys. Given a certain manufacturer, if a distributor performs its first delivery of an update to an IoT device, its score is instantiated with value 1. For any subsequent delivery, its score is incremented. This value, stored on the blockchain and accessible by anyone, can be used by other participants to judge the relative trustworthiness of any distributor compared to others. Additionally, the score of a distributor is periodically reset to 0 by the SSC, with a frequency that is decided by the manufacturer at the time of SSC deployment. This last mechanism avoids a situation in which certain distributors accumulate very high scores, making it very difficult for new distributors (starting with score 0) to be trusted by other participants.

Once the SSC is deployed, the manufacturer can proceed. In the rest of this chapter, we will focus on the actions performed by the following actors:

- The manufacturer m who is about to release a new update file U
- An IoT device o_m, one of the many manufactured by m and targeted by the release of U
- The hub h managing o_m, i.e. connected to the same local network
- A first-hand distributor d_f and a second-hand distributor d_s

The result of the steps described above will be threefold. (i) The IoT object o_m will have received the update file U. (ii) The distributor responsible for the delivery of the update to o_m will have received a cryptocurrency payment as a reward. (iii) And finally, the hub h will have received a cryptocurrency reward as well.

4.2.2 Update Release

The next step is for the manufacturer m to prepare a series of elements. Firstly, a pair of zk-SNARKs keys, the proving key pk^D and the verifying key vk^D. These are generated based on the following statement S_D, where the secret variables are U and r:

$$s = H(r) \wedge U_h = H(U) \wedge U_e = Enc(U, r)$$

In this statement, U is the new update to be released by m; U_h is its hash value, obtained applying the hashing algorithm H; U_e is an encrypted version of the update, obtained applying a symmetric encryption algorithm Enc and employing a key r; and finally s is the hash value of the key r. The manufacturer also needs to generate a second pair of zk-SNARKs keys, pk^E and vk^E, this time for the statement S_E with secret variables P and r:

$$s = H(r) \wedge P_h = H(P) \wedge P_e = Enc(P, r)$$

This other statement S_E is almost identical to S_D, except for the variable U replaced by P. The latter refers to another file, a package containing several elements which will be prepared by m in the very next steps. The purpose of these key pairs will be illustrated in Sections 4.2.3 and 4.2.4 respectively. To produce them, as explained in Section 2.3, m needs to have knowledge about two aspects of each variable in the statements: their size (not their actual values) and the specific algorithms used. For example, r could be a 256 bits key, and the manufacturer would not need to know its actual binary value. And the hash function H could be implemented with the SHA256 algorithm. S_D has almost the same structure as S_E except for the size of U compared to P: this is the reason why two different key pairs are needed.

Another element that m needs to prepare is the package P, the actual file that was referenced by the homonym variable in statement S_E. This package is constructed by m to contain (i) the update file U, (ii) the proving key pk^D, (iii) the verifying key vk^D and (iv) a signature generated by the manufacturer m upon the update hash value U_h, formally defined as $sig_m \coloneqq Sign_{prv^m}(U_h)$. This is the package that will be sent to distributors in the next stage of the protocol. Each internal element has its purpose. The one of the update U is obvious: this is the file that needs to be distributed to the target IoT devices. The zk-SNARKs keys will be used to generate and verify proofs throughout the protocol. And the signature sig_m is used by each IoT device to validate the update file in the very last steps of the protocol, avoiding the risk of accepting malicious updates.

Now the manufacturer m is ready to send a transaction to the SSC to trigger the creation of a new SC. In general, the SSC is able to generate two kind of derivate smart contracts, *delivery smart contracts* (DSCs) and *exchange smart contracts* (ESCs), depending on which SSC function is addressed by the triggering transaction. We will discuss ESCs in 4.2.3. Instead, in this case the manufacturer m generates a new DSC. In few words, this is the smart contract that will take care of issuing cryptocurrency rewards to distributors who are able to provide evidence that they delivered the update to the target IoT device. To deploy it, m needs to attach an arbitrary amount of cryptocurrency as a deposit and several other parameters. The deposit will serve as a source for the cryptocurrency rewards. The parameters are needed to initialize the state of the new DSC. It is worth noting that m is not required to attach the contract code, which would be necessary with traditional contract creation. Instead, the code is attached by m only once when the SSC is created, and then used as a template for new DSCs. This is convenient for m. But most importantly for other participants, as they can avoid expensive security checks on each new contract code. It is enough for them to concentrate on the security of a single SSC, to be assured that any DSC derived from it is secure as a consequence. The same reasoning is valid for ESCs. The parameters sent by m are the following:

- An integer value t_e indicating the time interval (e.g. measured in weeks) after which the DSC is considered to be expired.
- The hash values $U_h \coloneqq H(U)$, $P_h \coloneqq H(P)$, $vk_h^D \coloneqq H(vk^D)$, $pk_h^E \coloneqq H(pk^E)$ and $vk_h^E \coloneqq H(vk^E)$. These values will be part of the contract state, and therefore they will all be published on the blockchain. They can be used for integrity checks, exploiting the security features of the blockchain itself. That is, these values will be immutable and authenticated by the contract creator, which is the manufacturer. As a consequence, anyone can check the authenticity of a certain file by computing its hash value and comparing it with the corresponding value in the blockchain.
- The list L_m of PKs belonging to the target IoT objects. Among the others, this list contains the key pub^{o_m} belonging to the IoT object we are focusing on in this illustration.
- The values a_d and a_h representing the amounts of cryptocurrency to be sent for each reward, for distributors and hubs respectively.

When triggered for the creation of a new DSC, the SSC performs one single check on the received elements: the selected amounts a_d and a_h must be compatible with (i) the cryptocurrency deposit and (ii) the number of target IoT objects in the list L_m. This check is needed to avoid the case in which the DSC has not enough cryptocurrency balance to fund all the payments, which would not be fair to distributors who make an effort to deliver the update.

If this check is successful, a new DSC is deployed with a code that translates to the following algorithm. If any distributor with public key pub^d is able to present a valid proof-of-delivery (PoD) for any of the IoT targets in L_m, and no previous PoD was presented for that target before, then a cryptocurrency payment (of amount a_d) is sent to pub^d. Furthermore, if any hub with public key pub^h provides a valid proof-of-final-delivery (PoFD) for any of the targets, and no previous PoFD was previously presented for that target, a cryptocurrency payment (of amount a_h) is sent to pub^h. The previous actions can only be performed if the DSC is not expired, i.e. if the amount of time passed since its creation is still less than t_e. The nature of PoDs and PoFDs will be explained while illustrating the next steps. In short, they are signatures generated by IoT targets on specific values, and they are able to securely prove that a certain step of the protocol was performed for the benefit of the corresponding IoT device. The DSC can easily verify these signatures using the public keys listed in L_m.

4.2.3 Initial Seed and Additional Sharing

At this point, the protocol expects distributors to be regularly monitoring the blockchain network. In this way, when the manufacturer m triggers the creation of a new DSC, they become aware of the new update release. The next step is for distributors to request the package P via the peer-to-peer file sharing network. This is possible because the manufacturer has previously announced the availability of the package through its hash value P_h. So distributors can request the package using the same hash value, which can be retrieved from the DSC on the blockchain.

Distributor-Distributor Exchange

The initial seeding phase, where the manufacturer shares the update file with distributors, lasts for a limited amount of time. At the end of this temporary stage, a finite number of distributors has obtained the package P. As explained in Section 4.2.2, we refer to these kind of distributors as *first-hand* distributors (FHDs). Now, these FHDs compete against each other, trying to be the first to deliver the update to as many IoT devices as possible. Hence, they have no interest in sharing the package P with any new distributor willing to participate in the protocol. We call these new distributors *second-hand* distributors (SHDs). To allow them to participate, we introduced a fundamental feature to the *CrowdPatching* protocol. That is, a way for a SHD to obtain P from a FHD, in exchange for a cryptocurrency payment. Let us follow the actions taken by a SHD d_s to perform such exchange, with a FHD d_f who is in possession of P. In general, we refer to this interaction as a *distributor-distributor exchange* (DDE).

The SHD d_s sends a request for P on the P2P network, and d_f replies with an identification challenge c. To prove its identity, d_s sends back its public key pub^{d_s} along with a signature $sig_{d_s} \coloneqq Sign_{prv^{d_s}}(c)$. Once the signature is verified, d_f can check the score of d_s on the blockchain and decide whether to proceed with this interaction. This last steps is important for d_f to avoid wasting its resources. Indeed, the following steps include a zk-SNARKs proof generation and are therefore quite expensive in computational terms. We argue that, if the score associated with d_s is reasonably high, d_f can be quite confident about its intentions. In case d_f decides to trust d_s, the next step is to generate a zk-SNARK proof for the statement S_E illustrated in Section 4.2.2:

$$s = H(r) \wedge P_h = H(P) \wedge P_e = Enc(P, r)$$

Before doing so, d_f computes the following: (i) a fresh and random key t; (ii) the hash values $r \coloneqq H(t \,\|\, pub^{d_f})$ and $s \coloneqq H(r)$; (iii) the encrypted version of the package $P_e = Enc(P, r)$. Now the proof π can be generated using both the secret values and the public values for S_E as inputs, along with pk^E. The secret values are P and r. The non-secret values are P_h, P_e and s. This

proof is sent to d_s along with the public values that are not yet known to d_s itself, which are P_e and s. What is more, d_f also sends the keys pk^E and vk^E.

The keys can be verified by d_s using their hash values on the blockchain. Afterwards, d_s verifies the proof π using the public values P_h (retrieved from the blockchain), P_e and s, as well as vk^E. If the proof is valid, d_s is mathematically convinced about two important facts: (i) P_e was effectively obtained encrypting a package P with hash value P_h using a key r; (ii) s was indeed obtained computing the hash of r. In other words, d_s can now be sure to be in possession of an encrypted file P_e, which is the encryption of the exact package validated by the manufacturer through its hash value P_h on the blockchain. What is more, this encrypted file can be decrypted through a certain key r, which is still secret but has a non-secret hash value s. As a consequence, d_s only needs the hash pre-image of s to unlock the plaintext package.

At this point, d_s can send a transaction to the SSC to deploy a new ESC, the second kind of smart contract that can be generated. An arbitrary amount of cryptocurrency must be attached as a deposit. The code for an ESC can be summarized as follows. If the ESC is not expired, a cryptocurrency payment will be issued to any sender with public key pub^{d_f} who is able to provide the values t and r so that $r = H(t \,\|\, pub^{d_f})$ and $s = H(r)$. When these conditions are met, the key r will also be published on the blockchain. Once the ESC is deployed, d_f can send a transaction attaching t and r, which would trigger the cryptocurrency payment to d_f itself. The key r can then be used by d_s to decrypt P_e and obtain P. From this moment on, there is no difference between d_f and d_s, they both have all necessary elements to deliver the update to IoT targets and obtain rewards. What is more, since d_s also received the proving key pk^E from d_f, it can also perform exchanges with new SHDs who want to acquire the update.

It is important to note how the possibility for DHE exchanges impacts the scalability of the *CrowdPatching* protocol. In the work of Leiba et al [9], there is no such option. In their protocol, there is no way for new distributors to join after the initial seeding phase has ended. As a consequence, the number of distributors is severely limited by that particular time window. Instead, in *CrowdPatching* the number of distributors can grow indefinitely. Furthermore, it can grow at any point in the course of a given instance of the protocol. In other words, DDE exchanges can take place at any time, as they are not tied with any specific step of the protocol.

4.2.4 Update Delivery

Let us continue with the actions of a distributor d, which can be either d_f or d_s, or in general anyone in possession of the package P. The hub h, responsible for the IoT object o_m, discovers about a new update by looking at the blockchain. As a consequence, h sends a request through the peer-to-peer network, requesting U though its hash value U_h, which can be found on the blockchain. Through this mechanism, the hub h can find any node who announced the possession of the same file by enlisting its hash value. Additionally, h sends a fresh nonce n_1 to the selected distributor. The request is received by d, which sends back its public key pub^d, a signature $sig_d \coloneqq Sign_{prv^d}(n_1)$ and an identification challenge c.

At this point, h can verify if the signature sig_d was indeed generated using the received public key pub^d. If valid, h checks the integer score corresponding to pub^d on the blockchain. If the latter is not satisfying, h can search for a more trustworthy distributor. Otherwise, c is forwarded through the local network to the managed IoT object o_m. The latter now must generate a fresh nonce n_2, and subsequently generate a signature $sig^{ID}_{o_m} = Sign_{prv^{o_m}}(c \,\|\, n_2)$. This signature, along with n_2, is sent back to h. The use of the additional nonce n_2 is crucial for the security of the protocol. In the system proposed by Leiba et al. [9], the IoT object generates a signature on the challenge c directly, without first combining it with a fresh nonce. However, as we discovered during the formal analysis of this protocol described in Section 5.2, this constitutes a critical vulnerability. Indeed, the challenge c can be constructed in a way that the IoT device will generate a signature that effectively unlocks the cryptocurrency payment for the distributor. Instead, the simple concatenation of a fresh nonce in our protocol solves the problem. The details of this vulnerability and the corresponding solution are illustrated in Section 5.2.3.

Now the hub h can forward the signature $sig^{ID}_{o_m}$ to d, along with the nonce n_2 and the public key of the object pub^{o_m}. The distributor d verifies that (i) the signature $sig^{ID}_{o_m}$ was effectively

generated using the private key corresponding to the public key pub^{o_m} and (ii) the public key itself belongs to the list L_m on the blockchain. Afterwards, d can prepare for the generation of a zk-SNARK proof for the statement S_D introduced in Section 4.2.2:

$$s = H(r) \wedge U_h = H(U) \wedge U_e = Enc(U, r)$$

A series of elements needs to be computed for this purpose: (i) a random key t; (ii) the hash values $r := H(t \,||\, pub^{o_m} \,||\, pub^d)$ and $s := H(r)$; (iii) the encryption of the update file $U_e := Enc(U, r)$. The proof π can then be obtained employing both the secret values (U and r) and the public values for the statement S_D as inputs, along with the proving key pk^D. Finally, d can send π to the hub h, along with U_e, s, vk^D and the signature sig_m.

The hub verifies the zk-SNARKs proof π using the public values of S_D and the verifying key vk^D. If valid, h is mathematically convinced that: (i) U_e was indeed obtained encrypting a file U with hash value U_h using a key r; (ii) the hash value of the key r is s. At this point, the situation is very similar to the one described in Section 4.2.3 where a second-hand distributor has verified the zk-SNARKs proof received from a first-hand distributor. In an analogous way, in this case the hub h can now be certain to have received an encrypted file U_e which is the encryption of the exact update file that was authorized by the manufacturer through its hash value U_h on the blockchain. Furthermore, the hub is convinced that this encrypted file can be decrypted using a key r, which is still secret but has a known hash value s.

Consequently, h can proceed and request a proof-of-delivery (PoD) to the IoT object o_m. Along with this request, the hub forwards the values U_h, s and sig_m. The latter was published with the DSC on the blockchain. In turn, o_m verifies sig_m. In this way, the IoT device is assured that its manufacturer has indeed authorized the update file corresponding to the hash value signed in sig_m. If the signature is valid, o_m sends back another signature to h. That is, a signature that constitutes the PoD, defined as $sig^{PoD}_{o_m} := Sign_{prv^{o_m}}(U_h \,\|\, s)$. This PoD signature is fundamental and powerful. It is a way to formally declare to have received the encrypted update with hash value U_h, which can be decrypted with a key that has hash value s. This formal declaration in the shape of a signature can then be used by the distributor to claim the reward. Once received by h, the signature $sig^{PoD}_{o_m}$ is forwarded to d.

4.2.5 Key Publication and Final Delivery

The distributor d first verifies the signature $sig^{PoD}_{o_m}$ against the public key pub^{o_m} received before. If valid, d can post a transaction to the DSC, attaching various elements: (i) the public key of the targeted object pub^{o_m}; (ii) the values t, r and s; (iii) the PoD signature $sig^{PoD}_{o_m}$. If the DSC is expired, it simply ignores the transaction. If not expired, it checks the validity of the submitted values by testing a series of conditions in the following order:

1. The public key pub^{o_m} must be present the list L_m, and was not already served by another distributor. This can be enforced through a simple flag.
2. The equality $r = H(t \,||\, pub^{o_m} \,||\, pub^{sender})$ must be satisfied. This ensures that the key r was created by the sender of the transaction.
3. The equality $s = H(r)$ must be satisfied. This ensures that the submitted value s is indeed the result of hashing the submitted key r.
4. The signature $sig^{PoD}_{o_m}$ must match the concatenation of the update hash U_h and submitted value s, and must have been generated through the private key corresponding to pub^{o_m}.

If all these conditions are met, a cryptocurrency payment of amount a_d is issued to pub^{sender}, which is pub^d in this case, and the key r is published on the blockchain.

At this point, two other important actions are performed by the DSC. First of all, it sets the flag mentioned before, to store the information that this IoT object has already received the update. In this way, all other attempts from other distributors to obtain the reward for the same IoT device will be blocked. The second action is to increment the score of distributor d,

incrementing the integer value associated with its public key on the blockchain. This is done automatically, sending a transaction to the parent contract, i.e. the SSC. The latter checks if this is the first delivery for this specific distributor: if yes, it creates a new entry for it, with value 1. Otherwise, it checks if the reset period is expired: in this case the score is reset to 1 for the existing entry. Finally, in the third case, the score is incremented. The integrity and security of these operations is enforced through internal blockchain mechanisms, as well as through the well-formedness of the code of both contracts.

Now the hub h can retrieve the key r from the blockchain and decrypt U_e to obtain U. The update file is finally sent to the IoT device o_m. The latter checks the integrity of the file by generating its hash value and comparing it with the hash value received before, which was also signed by the manufacturer. If no errors occur, the object o_m sends back a signature to the hub, formally defined as $sig_{o_m}^{PoFD} := Sign_{prv^{o_m}}(U_h \,||\, pub^h)$. This is the proof-of-final-delivery (PoFD), a signature which can be used the hub to claim a cryptocurrency reward. Indeed, the hub can send this signature to the DSC. The smart contract checks the following:

1. The public key pub^{o_m} is in the list L_m and was not yet served by another hub. Again, this is achieved through a simple flag on the contract state.
2. The signature $sig_{o_m}^{PoFD}$ must be valid. That is, it must match the concatenation of the update hash and the sender's public key, which is pub^h in this case.

If no errors occur, the DSC issues a payment to the hub through its public key. What is more, the contract updates the corresponding flag, to reflect the fact that the object o_m has been already served by a hub for the update with hash value U_h.

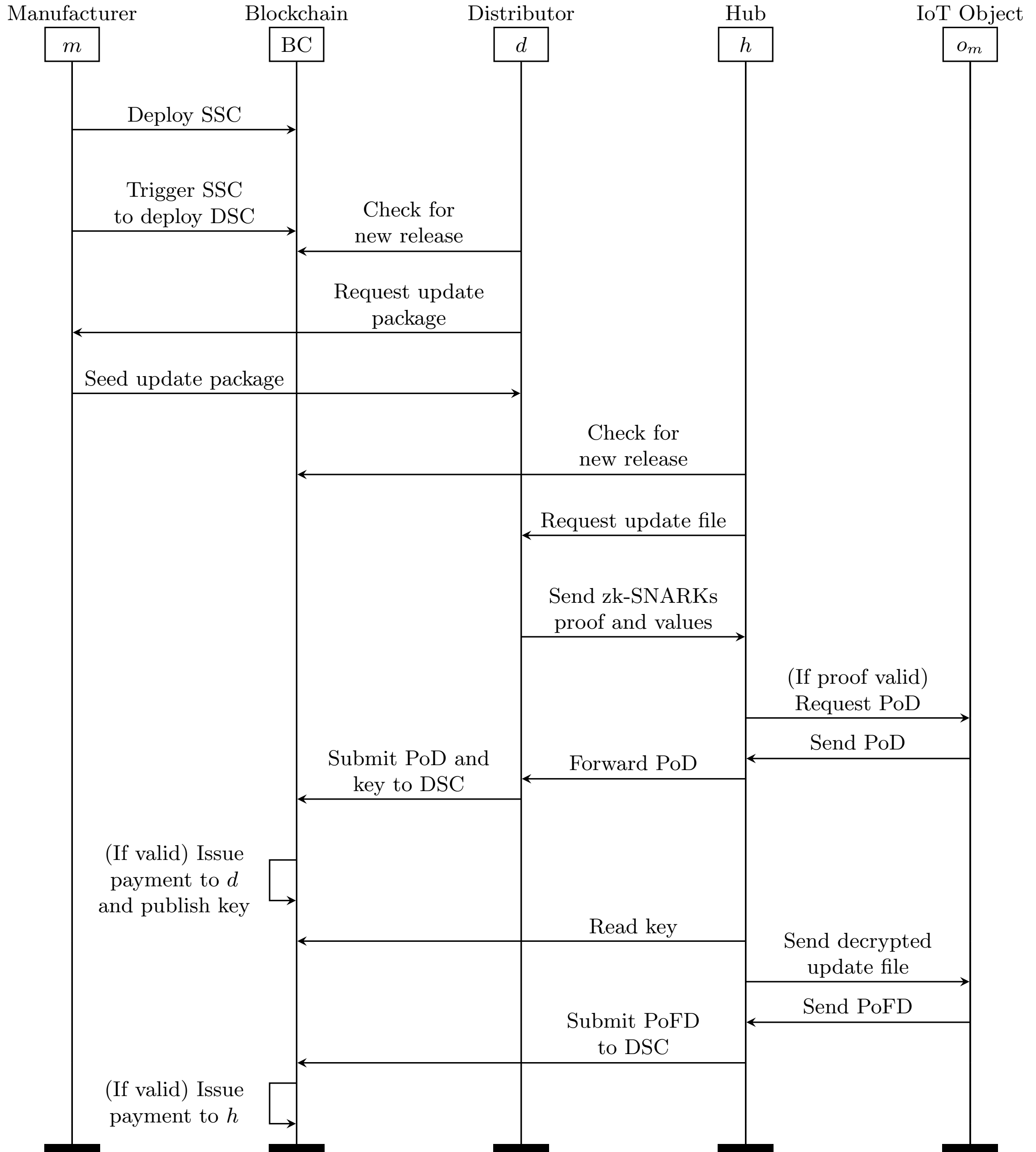

Figure 4.1. Overview of the protocol omitting the distributor-distributor exchange (DDE)

Chapter 5

Security Analysis

In this chapter, we present the security analysis of our protocol. We first provide an informal discussion of known attacks and vulnerabilities in Section 5.1. Subsequently, we illustrate the formal analysis we performed on the *CrowdPatching* protocol, by means of a state-of-the-art automated analysis tool called Tamarin. In particular, we introduce the Tamarin Prover in Section 5.2.1. We then continue in Section 5.2.2 illustrating the specific Tamarin model we designed, concluding with the security properties we successfully verified in Section 5.2.3.

5.1 Informal Analysis

5.1.1 Impersonation Attacks

We consider several scenarios in which an attacker attempts to impersonate legitimate entities of the protocol, in order to achieve malicious goals. We begin considering the case of an adversary masquerading as a manufacturer. The goal of the attacker could be to first deploy an SSC smart contract, and to subsequently trigger the SSC to deploy a DSC smart contract, which in turn would convince distributors to deliver a malicious update. In other words, the ultimate objective of the adversary would be to trick a number of IoT devices into accepting a malicious update. However, this is not possible for many reasons. First of all, the attacker would need to provide a reasonably conspicuous amount of cryptocurrency to fund the DSC, in order to incentivize distributors and hubs. We argue that this limitation alone would be enough to discourage this kind of attacks. On the other hand, an attacker could be motivated enough and decide to invest the needed amount of cryptocurrency to achieve this malicious objective. Even better, the adversary could decide to impersonate distributors at the same time, in order to partially obtain back the cryptocurrency rewards. Still, this attack would be unsuccessful for two main reasons. The first is that it would be impossible to deploy an SSC that would be trusted by hubs or honest distributors to be owned by a legitimate manufacturer. Indeed, we assume that the public key of any manufacturer is securely announced, and therefore known to hubs and honest distributors. Additionally, any transaction on the blockchain must be authenticated by a digital signature. As a consequence, there is no way for an attacker to deploy such an SSC on the blockchain without the knowledge of the manufacturer's private key. The second reason is even more assuring. The protocol requires an IoT device to first authenticate the hash value of any new update, as described at the end of Section 4.2.4. This authentication is achieved through a signature, which must be generated by a manufacturer through its private key. IoT devices can then verify this signature because they are in possession of the corresponding manufacturer's public key. What is more, at very end of the protocol (Section 4.2.5) an IoT device is required to verify the actual update file. That is, to compute its hash and compare it with the hash value authenticated before. As a consequence, there is no way for an IoT device to accept a malicious update which was not authorized by the manufacturer. The only viable option for an attacker in this case, would be to gain knowledge of the manufacturer's private key. In this case, a mechanism of key revocation should be put in place. However, the design of such mechanism is out of scope for this thesis.

We continue analyzing the case of an attacker impersonating one or more distributors. This can be easily achieved. However, the protocol is intentionally designed to allow this possibility, i.e. to allow literally anyone to participate as a distributor. For this reason, the protocol makes no security assumptions whatsoever about distributors. Several protections are implemented as a consequence, and in Section 5.1.3 we will consider how they function against malicious distributors.

It is instead impossible for an adversary to masquerade as a hub. We assume these entities to be manually selected by the owner of a set of IoT devices. They are assumed to be installed in a local network through a secure user configuration. Indeed, hubs have access to a special communication channel to interact with the managed IoT devices. The details of such channel are out of scope for this work. However, it is reasonable to assume that it would be easily established, and that it would be extremely difficult for an attacker to gain access to it. As a consequence, an attacker could never impersonate a hub while engaging with an honest distributor. The latter would request a response to the identification challenge, as illustrated in Section 4.2.4, which must be generated by an IoT device with its private key. Without access to the local network, the attacker would not be able to obtain such response from an IoT object, and the distributor would drop the communication. For the same reasons, the attacker would not be able to masquerade as a hub to communicate with an IoT device, at any moment.

Finally, it is impossible for an attacker to impersonate an IoT device without knowing its private key. The latter is embedded in each device by the manufacturer.

5.1.2 Interception of the PoD Submission

We consider the case of an attacker intercepting the submission of a proof-of-delivery (PoD) to the DSC on the blockchain, i.e. the smart contract issuing cryptocurrency payments as rewards for distributors and hubs. This submission is performed by a distributor in order to obtain the cryptocurrency reward, as explained in Section 4.2.5. In particular, a distributor can proceed with this submission if it successfully delivered the encrypted update to an IoT device o_m, which in turn generated the PoD signature $sig_{o_m}^{PoD} := Sign_{prv^{o_m}}(U_h \,\|\, s)$. Consequently, the elements sent by the distributor to the SSC include the signature $sig_{o_m}^{PoD}$ itself, as well as the values and pub^{o_m}.

Now, an attacker can easily intercept all these values due to the nature of the blockchain network. Any blockchain node issuing a new transaction is expected to broadcast the transaction itself, which includes all internal values in plaintext, to other nodes in the network. The transaction is then spread further in the network, so that it can reach miners. As explained in Section 2.1, these are special nodes gathering new transactions, and validating them to form new blocks that will be added to the main blockchain data structure. In this specific case, the distributor issues a new transaction addressing the DSC smart contract, and its internal values can then be eavesdropped by an adversary. The latter could attempt to block or replay these values to the DSC, in order to obtain the cryptocurrency reward in place of the honest distributor.

However, the protocol is well protected against this threat. Before issuing the payment, the DSC checks two equations to be satisfied, $r = H(t \,\|\, pub^{o_m} \,\|\, pub^{sender})$ and $s = H(r)$. In the first equation, pub^{sender} is the public key of the blockchain node issuing the transaction, which is a malicious node in this case. It is indeed trivial for the adversary to generate a new value for t, and then craft r and s in such a way that the equations would hold. But the attack would be stopped by a consequent verification performed by the DSC. That is, the validity of the signature $sig_{o_m}^{PoD}$. Given its definition above, the purpose of this signature is exactly to witness the authorization, made by the IoT device o_m, upon the value s. And this authorization is automatically extended to the value r, which must be the hash preimage of s, and to the value t, which must be the hash preimage of r. If we assume digital signatures to be cryptographically secure, an attacker can never manage to forge a PoD submission in its own favor.

Finally, an adversary who managed to intercept a PoD submission sent by an honest distributor d could try to replay its values exactly as they are. More precisely, it could try to submit the PoD to the DSC before the distributor, in order to claim the reward in advance. However, both the blockchain network internal mechanisms and the logic of the DSC would make this attack impossible. Indeed, the attacker would be forced to generate a signature on the submitted

transaction before issuing it. And the public key used to generate that signature would be seen by the smart contract as the pub^{sender}. However, the first of the two equations mentioned above would then be violated, because the value r was constructed by d to account for its public key pub^d and not the attacker's public key, formally as $r \coloneqq H(t \,||\, pub^{o_m} \,||\, pub^d)$.

5.1.3 Malicious Distributors

We consider the case of an attacker initially acting as a honest distributor, obtaining the update file from the manufacturer. The attacker could then interact with hubs with malicious intentions. In particular, the malicious distributor could waste the resources of a hub, producing a valid zk-SNARKs proof and then intentionally avoiding to submit the PoD to the DSC. As a consequence, the hub would not be able to decrypt the update file received along with the zk-SNARKs proof, and it would hopelessly wait for the distributor to send the decryption key to the DSC.

This threat is mitigated in two two ways. Firstly, the attacker would be strongly encouraged to submit the decryption key, because of the cryptocurrency payment that would immediately follow. Generating zk-SNARKs proofs is quite expensive in computational terms, which makes this attack quite disadvantageous if compared with the advantages of behaving honestly.

The other mitigation is the possibility for hubs to consult the score of any distributor. As described at the beginning of Section 4.2.4, a distributor is required to produce a signature on a nonce value (generated by the hub) at the beginning of its interaction with the hub. What is more, the distributor must send its public key. If the signature is valid, the hub can then retrieve the integer value corresponding to the received public key, which can be found on the blockchain as part of the DSC state variables. If the score is too low, the hub can simply drop the connection and look for more trustworthy distributors in the peer-to-peer file sharing network. We argue that this mechanism allows hubs to easily recognize malicious distributors, which would present very low scores. Most importantly, this second mitigation works as an incentive as well. That is, it encourages distributors to always complete their deliveries, because they will be reflected in their score on the blockchain.

Another possibility for an attacker acting as a distributor might be to attempt to generate a zk-SNARKs proof for a fake update. The goal of such an attacker would be to trick a hub, and its managed IoT device, into generating a PoD signature without having to deliver the real update file. The motivation could be that the attacker was not able to obtain the update file from the manufacturer in the initial phase, but still wants to get the cryptocurrency rewards for it without having to acquire it from other distributors. To achieve this, the attacker would need to generate a valid zk-SNARKs proof for the following statement, where F is an arbitrary file:

$$s = H(r) \wedge F_h = H(F) \wedge F_e = Enc(F, r)$$

In this statement, the secret values would be the file F and the key r, while the public values would be F_h, F_e and s. It would indeed be trivial for the attacker to generate a valid proof for this statement: it just needs to choose the arbitrary file F and the key r, compute the other values accordingly and then execute the zk-SNARKs *Prove* algorithm. However, the proof would not be accepted by the hub for two reasons.

First, if the attacker was not able to get the update from the manufacturer, then it was not able to obtain the update package, which also contains the proving key pk^D and the verifying key vk^D generated by the manufacturer. Without these keys, the attacker could think about generating a new pair of keys for the statement. Then it could send the fake verifying key to the hub along with the zk-SNARKs proof, as instructed by the protocol. But this key would not be accepted by the hub. As illustrated in Section 4.2.2, among the many values published by the manufacturer in the DSC smart contract, there the hash value of the verifying key, vk^D_h. After receiving the verifying key from the distributor, the hub is supposed to compute its hash and compare it with the value on the blockchain. For this reason, the fake key would be rejected.

Secondly, let us consider the case in which the attacker did obtain the update package, but still wants to generate a zk-SNARKs proof for an arbitrary file F. Along with this proof, the attacker sends the real verifying key vk^D and the public values F_e and s. This proof would still

be considered invalid by the hub. That is because the protocol does not instruct a distributor to send the third public value for the statement, in this case F_h. Instead, this public value is retrieved by the hub from the DSC on the blockchain. As a consequence, the *Verify* algorithm executed by the hub will judge the proof as invalid.

5.1.4 Update Integrity

We consider the case in which an attacker attempts to deliver a modified update, with the goal of inducing an IoT device to execute malicious code. One of them was discussed in Section 5.1.3: the attacker would need to generate a zk-SNARKs proof, which will be considered valid only if the encrypted value matches the hash value published on the blockchain by the manufacturer. The only other way for the adversary to successfully deliver a malicious update is to compromise a hub gateway. However, in this case the protocol remains secure. Before signing the identification challenge, an IoT device must verify a signature made by the manufacturer on the hash value of the update. Later in the very last step of the protocol, upon receiving the update file, the IoT device will check if the file matches the authenticated hash value. In this way, it is impossible for such devices to accept a malicious update unless (i) the device itself is compromised or (ii) the private key of the manufacturer was stolen by the attacker and used to generate a fake signature on the malicious file hash value.

5.1.5 Old Update Delivery

As explained in Section 5.1.4, there is no scenario in which an IoT device could accept a malicious update file. However, an attacker could attempt to deliver an old update. That is, an update that is not malicious per se, but can contain vulnerabilities that were fixed with newer updates. We consider two cases. In the first, we assume honest hubs. These gateway devices have access to the blockchain and they can see if a distributor is employing old values corresponding to an old update release. As a consequence, even if the values would be cryptographically acceptable, they would be rejected by a hub for chronological reasons. Indeed, the blockchain stores an immutable record of past releases, providing hubs with the perfect protection mechanism for this type of attack. In the second scenario, let us consider a compromised hub attempting to deliver an old update to the managed IoT device. This can be easily mitigated by the IoT device by keeping a record of previously received updates, e.g. storing their hash values. When a new update is received from the hub, the record is checked. If the update corresponds to any entry, it is rejected.

5.2 Formal Analysis

5.2.1 The Tamarin Prover

In this section we provide an introduction to the Tamarin Prover. That is, a protocol verification tool supporting the automated analysis of cryptographic protocols in the symbolic model [25, 26]. First of all, the Tamarin model of any protocol is defined in a specific file called *theory file*, with extension `.spthy`, which contains the entire definition of the symbolic protocol model as well as its security properties. The Tamarin Prover will then take this theory file as input, and run its automated analysis to verify the defined properties.

```
1 theory TheoryName
2 begin
3 builtins: hashing, asymmetric-encryption, symmetric-encryption, signing
4 ...
5 end
```

Listing 5.1. Structure of a Tamarin theory file

In general, a theory file has the structure reported in Listing 5.1. The `theory` keyword is used to declare the name of the protocol model, while all the remaining code is entirely included in a `begin` ... `end` construct. One can also optionally declare the usage of a subset of built-in theories. In other words, it is possible to employ default keywords that represent typical cryptographic tools. This is done through the `builtins` keyword placed right after the `begin` keyword. For example, in Listing 5.1, we imported the built-in theory definitions for hashing and signing algorithms, as well as for asymmetric and symmetric encryption. In the following, we provide a description of the main elements composing a Tamarin theory file within the `begin` ... `end` construct. In particular, we focus on the elements that are relevant to the model of our proposed protocol, which is presented in Section 5.2.2. What is more, we explain how a theory file is actually processed by Tamarin to prove, or disprove, its security properties. We refer to the Tamarin Official Manual [27] for more details on how to build theory files and prove their security properties.

Terms

In the symbolic model employed by Tamarin, messages are represented by terms. For example, a plain-text message and a cryptographic key could be represented by the simple variables `m` and `k`. Additionally, terms can correspond to constants or functions. An example of a function could be the built-in function for symmetric encryption. The expression `enc(m, k)` would be a term indicating the encryption of a message variable `m`, encrypted with the key `k`.

Cryptographic properties

The properties of the cryptographic functions are defined through equations over terms. For example, the built-in symmetric encryption algorithms are defined through the functions `enc` and `dec`. Their properties are defined through a single equation: `dec(enc(m,k),k) = m`, where `m` is the plaintext and `k` is the encryption key. This means that cryptographic primitives are effectively specified with a black-box approach. In other words, there is no need to indicate the inner workings of complex cryptographic algorithms. Instead, it is enough to specify their inputs and outputs, and the mathematical relationships between the various functions.

Many built-in cryptographic primitives are provided in the Tamarin language by default. They can be imported at the beginning of a theory file, as we mentioned when describing Listing 5.1. However, Tamarin also allows the user to create custom definitions.

Rules

A protocol model in Tamarin is defined through a labelled transition system, which in turn is defined by a collection of *multiset rewriting rules*. These rules have a specific structure, as shown in Listing 5.2. They have a left-hand side and a right-hand side, linked by an arrow that goes from left to right. Both sides are composed of an arbitrary number of facts. Optionally, a special set of facts called action facts can be indicated inside the arrow of the rule. In short, facts are used to model all the possible components of the system states, while rules describe the transitions between those states.

```
rule exampleRule:
  [ LeftFactX(term1, term2, ...), LeftFactY(...) ]
--[ ActionFactX(...), ... ]->
  [ RightFactX(...), ..., LeftFactY(...) ]
```

Listing 5.2. Structure of a Tamarin rule

The facts contained in the left-hand side and in the right-hand side are also called state facts, because they represent the state of the system before and after the state transition corresponding to the rule. At any moment during an instance of the protocol, the state of the system is represented by a collection of facts. A rule is triggered when the current state of the system

includes the facts that are indicated in the left-hand side of the rule itself. If this happens, the execution of the rule will *consume* this subset of facts, eliminating them from the state of the system. In turn, the right-hand side facts will be added to the state. Furthermore, if the rule contains action facts, they will be added to the execution trace. That is, a record of all the events triggered by all the rules executed in an instance of the protocol.

Facts

In general, a fact has a specific structure: `Fact(t1, t2, ... , tN)`. The fact symbol identifying the fact itself is `Fact`, while the symbols `ti` are the terms defining the fact. And the number of terms belonging to a certain fact is fixed, which means that the same fact must always appear with the same number of terms. A fact declared as above is defined as a *linear* fact by default. Linear facts are consumed by rules, when they are present in the left-hand side. As a consequence, if a linear fact appears in the left-hand side of a rule, the only way to maintain that fact in the state of the system when the rule is executed is to place it in the right-hand side too. For example, in the rule illustrated in Listing 5.2, all state facts are linear. When this rule is executed, the fact `LeftFactX` is eliminated from the state of the system. However, the fact `LeftFactY` is not consumed, because it appears in both the left-hand and right-hand sides.

Another type of fact exists, called *persistent* fact. A linear fact becomes a persistent fact if an exclamation point is added at its beginning: `!Fact(t1, t2, ... , tN)`. Persistent facts are never removed from the state of the system, even if they appear in the left-hand side of a rule that is executed while they are absent from the right-hand side. For example, referring the rule in Listing 5.2, if the fact `LeftFactX(...)` were to be defined as `!LeftFactX(...)` instead, then it would not be consumed by the execution of the rule.

In Tamarin, network communications are modeled with two special facts: the `In` fact and the `Out` fact. The former can be placed in the left-hand side of a rule and represents the reception of a message from the network. The latter can be placed in the right-hand side and represents the act of sending out a message to the network.

Another special fact is used to generate fresh and random values. That is, the `Fr` fact, allowing for example to generate a fresh nonce `n` by writing `Fr(~n)`. This special fact `Fr` can only be placed in the left-hand side of a rule. In general, the symbol `~` is a prefix indicating the fresh nature of the subsequent variable, and can also be used outside of the `Fr` fact.

Adversary Model

By default, Tamarin models the attacker as a Dolev-Yao adversary [28]. This type of attacker controls the entire network, but at the same time it is incapable of breaking cryptographic functions. As a consequence, it can manipulate messages sent in the network in any possible way, for example intercepting them or modifying them. However, given that cryptographic functions are assumed to be secure, the attacker cannot, for example, decrypt a message without the knowledge if the secret key used to encrypt it. And this knowledge can only be obtained if a message containing the key is sent through the attacker-controlled network. On the other hand, the attacker is allowed to execute all possible functions, such as the encryption built-in function.

Restrictions

Restrictions are used to limit the number of traces analyzed by Tamarin. As we will explain later, Tamarin verifies the properties of a protocol by executing all its possible instantiations, trying to find a protocol instance where the property is violated. However, the number of possible traces could easily become overwhelming even for simple protocols. For this reasons, there is a way to restrict this number: defining restrictions. These are special properties that must be valid for all traces. They are defined over action facts, i.e. facts that are indicated inside the arrow of a rule.

A typical example of a restriction is the `Equality` restriction, as shown in Listing 5.3. Here we also defined a rule where the linear facts `LeftFact1(x1)` and `LeftFact2(x2)` are consumed, while

the fact `RightFact(...)` is generated in the new state of the system. The rule presents an action fact `Eq(x1, x2)` that will be addressed by the restriction. The latter is defined as a mathematical property and essentially says the following: whenever a rule embedding an action fact `Eq(x, y)` is executed, it must be that `x` equals `y`. As a consequence, this restriction will apply to the above rule, which will be considered and executed by the Tamarin Prover only if the terms `x1` and `x2` included in the `Eq` fact are equal.

```
rule anotherExampleRule:
  [ LeftFact1(x1), LeftFact2(x2) ]
--[ Eq(x1, x2), EventActionFact(x1, x2, r) ]->
  [ RightFact(r) ]

restriction Equality:
  " All x y #i . Eq(x,y) @ #i ==> x = y "
```

Listing 5.3. Tamarin restriction example

Lemmas

Once all protocol rules and restrictions are defined, one can proceed to the specification of the security properties that Tamarin will attempt to prove. These properties are called *lemmas*. They are defined similarly to restrictions, but there is an important difference: the fact that lemmas are not enforced in a protocol execution. Instead, a property represented by a lemma needs to be explicitly verified, and can either be proved or disproved.

There are two types of lemmas. One is marked by the `exists-trace` keyword as in Listing 5.4 where the lemma `executabilityLemma` is defined. In general, this type of lemma is used to define executability properties. This is the kind of property that should be verified first for any given protocol model, as it makes sure that the protocol itself is executable in its entirety. Otherwise, if a protocol is not executable, other security properties could be falsely verified just because the steps of the protocol that would disprove them are not taking place. The `executabilityLemma` in Listing 5.4, if combined with the rule in Listing 5.3, applies this principle. More specifically, it makes sure that there exists a trace of the protocol in which the action fact `EventActionFact` is executed, proving the executability of this toy protocol made up of a single rule.

```
lemma executabilityLemma:
exists-trace
  " Ex x1 x2 r #i . EventActionFact(x1, x2, r) @i "
```

Listing 5.4. Tamarin `exists-trace` lemma example

The second type of lemmas is indicated with the `all-traces` keyword. However, this keyword can be omitted, because lemmas that do not indicate any of such keywords are considered as `all-traces` lemmas by default. The important difference with respect to `exists-trace` lemmas is that `all-traces` lemmas are verified only if the corresponding property holds for all possible traces of the protocol. For example, let as consider the lemma in Listing 5.5, which is identical to the one in Listing 5.4 except for the `all-traces` keyword in place of `exists-trace`. This means that if the `allTracesLemma` is verified, a much stronger property holds. That is, the fact that in all possible traces of the protocol the action fact `EventActionFact` is executed.

```
lemma allTracesLemma:
all-traces
  " Ex x1 x2 r #i . EventActionFact(x1, x2, r) @i "
```

Listing 5.5. Tamarin `all-traces` lemma example

Properties Verification

Finally, once a theory file is complete, the Tamarin Prover can be executed on the file itself. At this point, the user can explicitly verify the validity of any lemma. To do this, Tamarin explores all the possible protocol instances, analyzing all rules in a backward order.

There are two possible approaches depending on the type of lemma. For `exists-trace` lemmas, Tamarin will look for any instance of the protocol in which the property is verified. Three outcomes can result from this process. (*i*) If such instance is found, the process terminates and the proof witnessing the validity of the property is presented to the user. (*ii*) On the other hand, if all traces are explored without finding any instance in which the property is verified, then the lemma is considered to be disproven, i.e. the specified property does not hold. (*iii*) It can also happen that the process never terminates, if there is an infinite number of scenarios to be analyzed.

For `all-traces` lemmas, the approach is analogous and yet different. The mathematical expression defining an `all-traces` lemma is negated. Afterwards, the Tamarin Prover will explore all possible protocol traces to find an instance in which the negated property holds. In this way, if such a trace is found, the lemma will be considered to be disproven. This means that Tamarin was able to find an attack that violates the security property, and the proof of this vulnerability is displayed for the benefit of the user. Otherwise, if no such trace is found, the lemma is considered to be valid, meaning that the security property holds for all possible instances of the protocol. Similarly to what happens with `exists-trace` lemmas, this process can also never terminate.

5.2.2 *CrowdPatching* Protocol Rules

We present here the symbolic model for the *CrowdPatching* protocol. The corresponding theory file is located in the `tamarin` folder of our repository [29] with name `crowdpatching.spthy`. It starts with the usual `theory` keyword for specifying the name of the protocol, followed by the `begin` ... `end` construct. Additionally, as shown in listing 5.6, within this construct we import the built-in cryptographic primitives that will be used throughout the model:

- The `hashing` built-in function is simply a function `h` accepting one element as input. By default, unary functions like this one are defined as one-way functions, which is exactly the property characterizing cryptographic hash functions. In other words, it is not possible to retrieve the hash preimage given its hash value.
- The `asymmetric-encryption` built-ins are the functions `aenc` and `adec`, both accepting two terms as arguments (a message and a key) and implementing encryption and decryption respectively. A function `pk` is used to compute the public key from the private key.
- Analogously, the `simmetric-encryption` functions are `senc` and `sdec`, but there is no equivalent to the third additional function in this case.
- Finally, digital signatures primitives are imported with the `signing` keywords. The function `sign` accepts a message and a private key as inputs, and outputs the corresponding signature. The `verify` function takes a signature, a message and a public key as input and outputs the built-in value `true` if the signature is valid. A built-in equation is used to enforce this behavior: `verify(sign(m, sk), m, pk(sk)) = true`.

```
1 theory CrowdPatching
2 begin
3 builtins: hashing, asymmetric-encryption, symmetric-encryption, signing
4 ...
5 end
```

Listing 5.6. *CrowdPatching* theory file

Continuing with the content of our theory file, we introduce our custom definitions for the zk-SNARKs proving system. The Tamarin Prover does not provide these primitives by default,

so we had to design them. In general, Tamarin allows for the definition of arbitrary functions and equations through the `functions` and `equations` keywords, as shown in Listing 5.7. We defined the zk-SNARKs Tamarin model to account for the S_D statement only. That is, the zk-SNARKs statement illustrated in Section 4.2.2 that is used by distributors to generate zk-SNARKs proofs for hubs. On the other hand, we did not model the second statement S_E. This is because, as explained later in this section, we modeled a simplified version of the *CrowdPatching* protocol. As a consequence, the protocol steps in which the statement S_E is involved are not modeled.

Functions are defined indicating their name and their arity, i.e. the number of arguments accepted as input. We firstly define two constant functions, `GenProvKey` and `GenVerifKey`, which model the proving key and the verifying key. A constant function is essentially just a constant, and is declared as a function with arity 0. Additionally, we defined these functions as `private`. This option allows to define functions that cannot be executed by the attacker, whereas all other functions can be freely exploited by the adversary. This was done to reflect the fact that, in the *CrowdPatching* protocol, these keys are securely generated by the manufacturer when the attacker cannot interfere. We also define the `zkProve` and `zkVerify` functions, with arities 2 and 3 respectively. The former represents the zk-SNARKs *Prove* algorithm, which takes secret and public values and the proving key. The latter models the *Verify* algorithm, with the proof, the non-secret values and the verifying key as inputs. Note that in both functions, public values and secret values are each treated as single terms. This is possible through a special syntax tool in Tamarin: you can have several elements (e.g. `a`, `b` and `c`) and enclose them between angle brackets (`<a, b, c>`) to treat them as a single term. Finally, we defined another constant called `ver`, which simply represents the output of the *Verify* algorithm when the processed proof is valid.

```
1 functions:
2   GenProvKey/0 [private], GenVerifKey/0 [private],
3   zkProve/2, zkVerify/3, ver/0
4
5 equations:
6   zkVerify( GenVerifKey, <h(U), senc(U, r), h(r)>,
7     zkProve( GenProvKey, << h(U), senc(U, r), h(r) >, <U, r>> ) ) = ver
```

Listing 5.7. Custom cryptographic functions for zk-SNARKs

In addition to these function, we defined a single equation to define their behavior and relationships with each other. As explained in Section 5.2.1, this is a black-box approach. Consequently, there is no need to specify the inner workings of the zk-SNARKs functions. Instead, it is enough to specify their possible outputs according to their inputs. In this case, we simply provided a definition of the scenario in which the `zkVerify` function accepts a zk-SNARKs proof as valid. More specifically, we built an equation. On the right side we have the `ver` constant, representing a successful output for `zkVerify` function. On the left side we have an instance of the `zkVerify` function with the following arguments as input:

1. The verifying key, i.e. the constant function `GenVerifKey`
2. The public (non-secret) values enclosed in angle brackets:
    2.1. The hash of the update file `h(U)`
    2.2. The encryption of the update file `senc(U, r)`
    2.3. The hash of the encryption key `h(r)`
3. An instance of the `zkProve` function with the following arguments:
    3.1. The same public values as before: `<h(U), senc(U, r), h(r)>`
    3.2. The secret values enclosed in angle brackets:
        3.2.1. The update file `U`
        3.2.2. The encryption key `r`

Another important element of our model is the public-key infrastructure (PKI). That is, a system able to provide all entities with their private-public key pairs. To achieve this, we followed the guidelines of the official Tamarin Manual, as shown in Listing 5.8. A single rule allows any entity to generate a fresh private key `~ltk` (line 2), which is then linked to a public identity variable `$X` (line 4). Additionally, the public key is derived from the private key and linked to the same identity (line 5). Finally, the public key is sent out to the network (line 6). There are no action facts in this rule, which is why the arrow is empty (line 3).

```
rule publicKeyInfrastructure:
  [ Fr(~ltk) ]
--[ ]->
  [ !Ltk($X, ~ltk)
  , !Pk( $X, pk(~ltk) )
  , Out( pk(~ltk) ) ]
```

Listing 5.8. Public-key infrastructure (PKI) model

Protocol Model Setup

We designed a model of a simplified version of the *CrowdPatching* protocol with a limited number of entities. The reason why we opted for this approach is the fact that a Tamarin model can easily have an overwhelming complexity, which would lead to the Tamarin verification engine to never terminate. As a consequence, we realized it would have been unfeasible to consider a realistic scenario with respect to the number of IoT devices. Instead, we designed a model in which a single manufacturer releases a single update targeting three IoT devices, along with other simplifications. In the theory file, this simplified model is initialized through a setup rule. That is, a Tamarin rule which does not correspond to any step of the protocol, with the sole purpose of instantiating all entities with their initial state. This essential rule is shown in Listing 5.9. We describe it in the following, along with the simplifications it reflects:

- We exploit `the let ... in` construct. That is, a way of defining macros that are valid in the current rule scope. We define five macros (lines 4-8). For example, as a consequence of this construct, the keyword `Uh` will always be replace by the expression `h(U)` in this rule.
- A single manufacturer `M` releases a single update, targeting three IoT devices `IoT1`, `IoT2` and `IoT3`. Two of this devices are managed by a certain hub, `H1`, while the other object is managed by a second hub `H2`. The number of distributors is limited to three: `D1`, `D2` and `D3`. The private keys of all these entities are initialized in the left-hand side of the setup rule (lines 11-15). The only other action performed in this side of the rule is to generate a fresh value `U` representing the update file (line 11).
- We avoid modeling the SSC smart contract, and we directly generate the DSC instead. In other words, we start our model assuming that the manufacturer has already deployed a DSC, the contract that takes care of issuing cryptocurrency payments to distributors and hubs as rewards. This is modeled through a series of facts (lines 19-33):
    - The permanent fact `!DSC_Info` allows any entity of the protocol to consult the public information associated with the DSC on the blockchain: the update hash, the signature by the manufacturer on the update hash, and the hash of the verifying key. All these elements are also sent out in the public network, to model the fact that anyone can access them, including the Tamarin adversary.
    - Three linear facts represent three independent initial states for the DSC, one for each IoT device. These facts have the same name, `St_DSC_0`, but different terms.
    - Three permanent facts with name `!DSC_Info_IoT`, one for each IoT device, represent the information about each device on the blockchain, to reflect the immutability property of this data structure.

  - Three `Out` facts are used to send out the same information to the network, to model the fact that anyone can access it on the blockchain.

- All remaining facts in the right-hand side of the rule represent the initial states of all entities. They all have the same structure, as they all represent the state of an entity, which is the state with index 0 in this case. In general, we use the following notation. The name of a fact representing the state i of an entity `X` is `St_X_i`, where `X` can be `IoT`, `H` or `D`. Its terms are two: the string identifying the specific instance of the entity, e.g. `'IoT1'` for the object `IoT1`; the elements composing the state enclosed in angle brackets, e.g. `<a, b, c>`. The actual state facts for the setup rule are as follows:

  - Three facts with the same name, `St_IoT_0` represent the initial state of the three IoT device. Each of this facts presents three state-components: (*i*) the private key of the object; (*ii*) the public key of the manufacturer; (*iii*) the identity of the associated hub. These facts have the same name for a specific reason. This allowed us to define the subsequent rules in a universal way, independent of the specific instance the IoT entity that will execute the rule itself. In other words, other rules in this model are designed for IoT devices regardless of the specific device, e.g. `IoT1` or `IoT2`.
  - The subsequent three facts with name `St_H_0` represent the initial state of the two hubs. They all have the same name, for analogous reasons as above. On the other hand, two of them have identity `H1`, the other `H2`. This is because `H1` manages two IoT objects, while `H1` only one. Each of them has two state-components: (*i*) the private key of the corresponding hub and (*ii*) the identity of the managed IoT object.
  - Finally, nine facts with name `St_D_0` initialize the state of three distributors. There are three facts for each distributor `D1`, `D2` and `D3`. This was done to model the fact that each distributor is independently interested in delivering the update to all three IoT devices, competing with the others. The state-components are: (*i*) the private key of the corresponding distributor; (*ii*) the package `P`, defined in the `let ... in` construct; (*iii*) the identity of the target IoT object. The state of each distributor contains the package `P` because we are omitting the initial seeding phase of the protocol. That is, we assume distributors to be have already obtained the update package from the manufacturer without modeling this passage.

- We also omit hub rewards: we assume hubs to behave honestly.

When this setup is executed, all facts in the right-hand side of the rule are added to the current state of the system. The action fact `UpdatePublished` is not relevant in this context. It is referenced by lemmas in order to define security properties relating to this rule. On the other hand, the `Setup` action fact is used within a restriction to make sure this rule is executed only once in every instance of the protocol.

Protocol Steps

After the setup rule, the theory file contains a long series of rules describing all steps of the protocol. Out of all possible approaches, we decided to embrace the following. We designed a rule for each step of each protocol role, where a role can be either the IoT role, the hub role or the distributor role. In other words, regardless of the specific identity that will take on a role (e.g. `IoT1`) we define a single rule for each protocol step corresponding to that role (e.g. `IoT`). The alternative could have been to mix different protocol roles in a single rule, but we avoided this option to make the theory file more readable. We could have also opted for as many rules as the number of identities for each step, but this would have brought no benefits to the design.

Furthermore, the rules are listed chronologically in the theory file. Rules for different roles are intertwined with each other, so that consequent rules in the theory file correspond to consequent steps of the protocol. The alternative would have been to group all rules related to the same role. This is not relevant for the Tamarin syntax: rules can be placed in any order as long as they are contained in the `begin ... end` construct.

```
rule setup:

  let
    Uh = h(~U)
    PK  = GenProvKey
    VK  = GenVerifKey
    P = <~U, PK, VK>
    sigByM = sign(Uh, ~ltkM)
  in

    [ !Ltk('M', ~ltkM), Fr(~U)

    , !Ltk('IoT1', ~ltkIoT1) , !Ltk('IoT2', ~ltkIoT2), !Ltk('IoT3', ~ltkIoT3)
    , !Ltk('D1', ~ltkD1), !Ltk('D2', ~ltkD2), !Ltk('D3', ~ltkD3)
    , !Ltk('H1', ~ltkH1), !Ltk('H2', ~ltkH2) ]

  --[ Setup(), UpdatePublished(~U) ]->

    [ !DSC_Info( Uh, sigByM, h(VK) )

    , Out( <pk(~ltkM), Uh, sigByM, h(VK)> )

    , St_DSC_0( 'DSC', <'IoT1', pk(~ltkIoT1)> )
    , St_DSC_0( 'DSC', <'IoT2', pk(~ltkIoT2)> )
    , St_DSC_0( 'DSC', <'IoT3', pk(~ltkIoT3)> )

    , !DSC_Info_IoT( 'IoT1', pk(~ltkIoT1) )
    , !DSC_Info_IoT( 'IoT2', pk(~ltkIoT2) )
    , !DSC_Info_IoT( 'IoT3', pk(~ltkIoT3) )

    , Out( <'IoT1', pk(~ltkIoT1)> )
    , Out( <'IoT2', pk(~ltkIoT2)> )
    , Out( <'IoT3', pk(~ltkIoT3)> )

    , St_IoT_0( 'IoT1', <~ltkIoT1, pk(~ltkM), 'H1'> )
    , St_IoT_0( 'IoT2', <~ltkIoT2, pk(~ltkM), 'H1'> )
    , St_IoT_0( 'IoT3', <~ltkIoT3, pk(~ltkM), 'H2'> )

    , St_H_0( 'H1', <~ltkH1, 'IoT1'> )
    , St_H_0( 'H1', <~ltkH1, 'IoT2'> )
    , St_H_0( 'H2', <~ltkH2, 'IoT3'> )

    , St_D_0( 'D1', <~ltkD1, P, 'IoT1'> )
    , St_D_0( 'D1', <~ltkD1, P, 'IoT2'> )
    , St_D_0( 'D1', <~ltkD1, P, 'IoT3'> )

    , St_D_0( 'D2', <~ltkD2, P, 'IoT1'> )
    , St_D_0( 'D2', <~ltkD2, P, 'IoT2'> )
    , St_D_0( 'D2', <~ltkD2, P, 'IoT3'> )

    , St_D_0( 'D3', <~ltkD3, P, 'IoT1'> )
    , St_D_0( 'D3', <~ltkD3, P, 'IoT2'> )
    , St_D_0( 'D3', <~ltkD3, P, 'IoT3'> ) ]
```

Listing 5.9. Setup rule for the *CrowdPatching* model

In the remaining of our theory file, there is a conspicuous number of rules describing all protocol steps. However, it would be unfeasible to report each one of them. Instead, we will present the most significant rules, corresponding to the most crucial steps of the protocol. The entire theory file can be found in our public repository online [29].

It is relevant to show the first rule after the setup. That is, the rule `H_1` modeling a hub requesting the update file through the network, as shown in Listing 5.10. In general, we name a rule as `X_i` when it models the transition of an entity `X` from state $i-1$ to state i. In this case, the hub entity is transitioning from state 0 to state 1. The former is represented by the fact `St_H_0` in the left-hand side, the latter by `St_H_1` on the right-hand side. This rule is triggered when the state of the system contains (i.e. includes) the facts `St_H_0` and `!DSC_Info` in the left-hand side. As a consequence, this rule is triggered right after the setup rule, which generates exactly those facts in the right-hand side and adds them (among many others) to the state. However, the fact `St_H_0` has different terms in the rule `H_1` compared to the setup rule: in this case they are variables that can assume different values depending on the actual entity that is executing this rule. For example, the first term of `St_H_0` is the variable `$H`. This can correspond to either `'H1'` or `'H2'`. Similar reasoning applies to the other terms. As a consequence, the fact `St_H_0` can match any of the three facts with the same name generated by the setup rule.

```
1  rule H_1:
2    [ St_H_0($H, <~ltkH, $IoT>)
3    , !DSC_Info(Uh, sigByM, VKh) ]
4  --[ ]->
5    [ Out(<'UpdateRequest', $IoT, $H>)
6    , St_H_1($H, <~ltkH, $IoT, Uh, sigByM, VKh>) ]
```

Listing 5.10. Hub requesting the update (Tamarin rule)

Once the `H_1` rule is triggered, the left-hand side facts are consumed, meaning that they are eliminated from the state of the system. However, the fact `!DSC_Info` is persistent, so it will not be deleted. It was placed here to account for the fact that the hub discovers about a new update release by monitoring the blockchain, waiting for a new DSC. Subsequently, the right-hand facts are added to the state. The new fact-state of the hub `St_H_1` has the same state-components plus the terms recovered from `!DSC_Info`. Most importantly, the `Out` fact sends out the update request to the network. It will be received by a distributor in a rule with the `In` fact in the left-hand side.

```
1  rule IoT_1:
2
3    let
4      result                     = verify(sigByM, Uh, pkM)
5      sigOnChallengeNonceByIoT   = sign( <c, ~nonce>, ~ltkIoT )
6    in
7
8    [ St_IoT_0( $IoT, <~ltkIoT, pkM, $H> )
9    , LocalChannel( $H, $IoT, <'IdChallenge', c, Uh, sigByM> )
10   , Fr(~nonce) ]
11
12 --[ Eq( result, true ) ]->
13
14   [ LocalChannel( $IoT, $H, <'IdReply', sigOnChallengeNonceByIoT, ~nonce> )
15   , St_IoT_1( $IoT, <~ltkIoT, pkM, $H, Uh> ) ]
```

Listing 5.11. IoT signing the ID challenge (Tamarin rule)

We omit the subsequent two rules. They model (i) a distributor receiving the update request and replying with an identification challenge and (ii) the hub receiving the challenge and forwarding it to the managed IoT object. The next step is modeled in the `IoT_1` rule, shown in Listing 5.11. Here an IoT object transitions from state 0 to state 1. In the left-hand side, the IoT device receives the ID challenge in the local channel established with the hub. The latter is represented

by a linear fact that was generated by the hub in the previous rule. In this way, the attacker cannot interfere as it could have done if the `Out` and `In` facts were used instead. Additionally, a fresh nonce is generated.

In the rule arrow we find an important action fact. That is, the `Eq` action fact enforcing the equality of the two terms included as arguments. The term `result` is defined in the `let ... in` binding as the result of the signature verification executed on the signature by the manufacturer. The `true` constant is built-in and indicates a successful outcome of any signature verification. In other words, we are imposing that this rule can be executed only if the signature is valid. However, the action fact is not enough to actually enforce this behavior. As explained in Section 5.2.1, it needs to be coupled with a specific restriction, shown in Listing 5.12. This restriction applies to any `Eq` action fact that is employed throughout the entire theory file.

```
restriction Equality:
" All x y #i. Eq(x,y) @i ==> x = y "
```

Listing 5.12. Equality restriction

In the right-hand side of the `IoT_1` rule, the IoT device generates a signature on the concatenation of the challenge and the fresh nonce. This is embedded in the definition of the `sigOnChallengeNonceByIoT` macro. The resulting signature is sent back to the hub through another `LocalChannel` fact. The new state-fact `St_IoT_1` is also instantiated, containing the new value `Uh`, which is the hash value that was just authenticated by the signature `sigByM`.

We omit the subsequent rule, where the hub simply forwards the `sigOnChallengeNonceByIoT` signature to the distributor through the attacker controlled network. We present the rule in which this signature is received by the distributor, shown in Listing 5.13 and called `D_2`.

```
rule D_2:

let
  P        = <U, PK, VK>
  result1 = verify(sigOnChallengeNonceByIoT, <~c, nonce>, pkIoT)
  r        = h(<~t, pkIoT, pk(~ltkD)>)
  Uenc     = senc(U, r)
  s        = h(r)
  sec      = <U, r>
  pub      = <h(U), Uenc, s>
  pi       = zkProve(PK, <pub, sec>)
  result2 = zkVerify(VK, pub, pi)
in

  [ St_D_1( $D, <~ltkD, P, $IoT, pkIoT, $H, ~c> )
  , In( <'IdReply', $H, $D, $IoT, sigOnChallengeNonceByIoT, nonce> )
  , Fr(~t) ]

--[ Eq(result1, true), Eq(result2, ver)
  , GenProof(pk(~ltkD), $IoT, U) ]->

  [ Out( <'zkSNARKsProof', $D, $H, $IoT, pi, Uenc, s, VK> )
  , St_D_2( $D, <~ltkD, P, $IoT, pkIoT, $H, ~t, r, s> ) ]
```

Listing 5.13. Distributor generates zk-SNARKs proof (Tamarin rule)

In the left-hand side of the rule, the signature is indeed received through the `In` fact. Thanks to the `Eq(result1, true)` action fact, the distributor executing this rule will not proceed unless the signature is verified. If that is the case, a fresh key `~t` is also generated in the left-hand side to be used in the right-hand side. This key is used to prepare the zk-SNARKs proof, a process which is mostly reflected in the `let ... in` construct. The key `r` is computed as the hash `h(<~t, pkIoT, pk(~ltkD)>)`. The update file is encrypted to obtain `Uenc` and the key `r` is hashed to obtain

`s`. And finally the zk-SNARKs proof `pi` is generated by executing the function corresponding to the *Prove* algorithm: `zkProve(PK, <pub, sec>)`. The generated proof is sent through the attacker-controlled network, directed to the hub who had requested the update. Other values are attached: the encrypted update `Uenc`, the key hash `s` and the verifying key `VK`. The action fact `GenProof` is used to record the generation of the zk-SNARKs proof when this rule is executed.

We omit the subsequent rules in which (*i*) the hub verifies the zk-SNARKs proof `pi` and forwards the value `s` to the IoT object; (*ii*) the IoT object generates the proof-of-delivery (PoD) signature and sends it to the hub; (*iii*) the latter is forwarded by the hub to the distributor through the network; (*iv*) the distributor submits the PoD to the smart contract.

At this point, the first and only rule modeling the DSC smart contract can be executed. That is, the `DSC_1` rule shown in Listing 5.14. Here the submission is received through the attacker-controlled network by means of the `In` fact. Defined in the `let ... in` construct, it contains all necessary values that are verified by the DSC with three equality action facts:

- `Eq(r, h(<t, pkIoT, pkD>))` verifies that the key `r` was obtained in Section 4.2.4
- `Eq(s, h(r))` verifies that `s` is the hash of `r`
- `Eq(verify(deliveryProof, <s, Uh>, pkIoT), true)` verifies that the PoD signature is valid

If these equalities are satisfied, the reward cryptocurrency payment is issued in the form of an action fact called `PaymentToD`. The latter serves as a record that the payment has been issued to the distributor with public key `pkD`, and can be addressed by lemmas to define security properties. Finally, the DSC publishes the decryption key on the blockchain. This is done by means of a permanent fact called `!DSC_Info_UpdateDecryptionKey`, which associates the identity of the involved IoT object with the key `r`.

```
rule DSC_1:

let
  submission = <pkD, $IoT, t, r, s, deliveryProof>
in

  [ St_DSC_0('DSC', <$IoT, pkIoT>)
  , !DSC_Info(Uh, sigByM, VKh)
  , In(<'DeliveryProofSubmission', submission>) ]

--[ Eq(r, h(<t, pkIoT, pkD>)), Eq(s, h(r))
  , Eq(verify(deliveryProof, <s, Uh>, pkIoT), true)
  , PaymentToD(pkD, $IoT) ]->

  [ !DSC_Info_UpdateDecryptionKey($IoT, <Uh, r>) ]
```

Listing 5.14. PoD validation by the DSC (Tamarin rule)

The final rule in Listing 5.15 models the actions of a hub retrieving the key `r` from the blockchain. This is done by placing the `!SC_Info_UpdateDecryptionKey` permanent fact in the left-hand side. By means of two equality action facts, this rule also checks that the key is indeed the hash pre-image of the value `s`, and that the hash of the decrypted update corresponds to the hash value obtained before from the blockchain. An action fact called `UpdateReadyForIoT` has the simple purpose of recording this event, associating it with the `$IoT` identity. This action fact will be referenced by lemmas to enforce specific security properties.

We avoided the construction of an additional rule that would simply model the hub sending the decrypted update to the managed IoT device. This is because, as mentioned at the beginning of this section, we do not model hub rewards in Tamarin. As a consequence, such an additional rule would be completely pointless, because it would only contain a communication through the IoT-hub local channel, which is secure by definition in this model.

```
rule H_6:

let
  Udec = sdec(Uenc, r)
in

  [ St_H_5($H, <~ltkH, $IoT, Uh, Uenc, s>)
  , !SC_Info_UpdateDecryptionKey($IoT, <Uh, r>) ]

--[ Eq(s, h(r)), Eq(Uh, h(Udec))
  , UpdateReadyForIoT($IoT, Udec) ]->

  [ St_H_6($H, <~ltkH, $IoT, Uh, pub>) ]
```

Listing 5.15. Update decryption by hub (Tamarin rule)

5.2.3 *CrowdPatching* Security Properties

We present here the security properties that have been successfully verified by the Tamarin Prover. More precisely, as explained in Section 5.2.1, the Tamarin software has the capability of injecting a theory file, analyzing its protocol rules and allowing the user to explicitly launch the verification of any lemma. This verification can be fully automated by the Tamarin engine, leveraging deduction, equational reasoning and heuristics. We verified all the following lemmas by means of this automated modality. However, it is worth noting that, in case the verification of a lemma leads to non-termination, Tamarin allows for other non-automated modalities.

Protocol Executability

Before verifying any protocol security property, it is fundamental to ensure that the model is executable. That is, to ensure that all its rules are well-formed and allow the entirety of the protocol steps to be executed. This is done through the `exist-trace` lemma shown in Listing 5.16, which is generally referred to as an executability lemma. The `exists-trace` keywords means that, when this lemma is processed by the Tamarin engine, it is considered to be verified if there exists a trace in which the property holds.

```
lemma ExecutabilityAllIoTGetUpdate: exists-trace
"
Ex #t0 #t1 #t2 #t3 U
  . UpdateReadyForIoT('IoT1', U) @t1
  & UpdateReadyForIoT('IoT2', U) @t2
  & UpdateReadyForIoT('IoT3', U) @t3
  & UpdatePublished(U) @t0
  & t0 < t1 & t0 < t2 & t0 < t3
"
```

Listing 5.16. Executability lemma for the *CrowdPatching* model

As for any lemma, the property is defined through a mathematical formula enclosed in double quotes. Traditional math symbols can be used. The symbol $\exists$ is represented by the `Ex` keyword, while the `All` keywords is used in place of the symbol $\forall$. Timepoints are indicated with the symbol `#` as a prefix when they are declared, whereas the symbol `@` is used when they are referred to after their declaration. The symbol `@` can also be omitted in some cases. A simple full stop symbol represents the "so that" mathematical symbol. Finally, the symbol `&` is used to put different conditions in logical conjunction with each other.

In this case, the formula associated with the `ExecutabilityAllIoTGetUpdate` lemma ensures that there exist four time points (`t0`, `t1`, `t2`, `t3`) and an update variable `U` so that all the following conditions apply: (i) the update is ready for each IoT device at independent timepoints (`t1`, `t2`,

t3); (*ii*) the update received by all devices is the same that was published (in the setup) at another timepoint (t0); (*iii*) the update was published at a previous timepoint compared to all timepoints in which the update was ready for the IoT devices. This lemma was indeed verified when it was processed by the Tamarin Prover, meaning that a protocol trace exists where all the protocol rules can be executed as described by the formula. In other words, the validity of this lemma proves that all rules can be successfully executed until, including the very last rule reported in Listing 5.15. What is more, since this is an exists-trace lemma, when Tamarin finds a valid trace it also generates a graphic representation reporting the whole execution of the protocol. Each rule is represented by a rectangle with three rows, one for the left-hand side, one for the action facts inside the arrow and one for the right-hand side. These rectangles are linked with arrows that draw a complicated flow of rules connecting with each others. The graphic representation generated for the ExecutabilityAllIoTGetUpdate lemma can be found in our online repository [29].

Protocol Fairness for IoT Objects

Once the executability of the protocol is verified, one can proceed with the verification of specific security properties. In our case, the first of these is encoded in the PaymentOnlyIfGenerateProof lemma shown in Listing 5.17. Differently from the executability lemma, this one employs the all-traces keyword. As a consequence, it can be considered valid only if the corresponding mathematical formula holds for all possible traces of the protocol.

```
lemma PaymentOnlyIfGenerateProof:
all-traces
"
  All #j pkD IoT
  . PaymentToD(pkD, IoT) @j

  ==> Ex #i U
      . GenProof(pkD, IoT, U) @i
      & i < j
"
```

Listing 5.17. *CrowdPatching* security lemma: protocol fairness for IoT objects

This lemma ensures that (in each trace of the protocol) whenever a PaymentToD(pkD, IoT) action fact is generated at a certain timepoint j, then it must be that a GenProof action fact referring to the same public key pkD and IoT device was generated at an earlier timepoint i. In other words, we are enforcing that whenever a cryptocurrency payment is issued (event recorded with the action fact PaymentToD) to a certain distributor (identified by its public key) then it must be that the same distributor generated a zk-SNARKs proof for the same IoT object associated with the reward. As a consequence, this ensures that a payment is issued only if a distributor has previously delivered the encrypted update to the IoT device.

The verification of this lemma allowed us to discover a vulnerability affecting an old version of our protocol. That is, a version in which the protocol steps illustrated in Section 4.2.4 were slightly different. In particular, in the old version the IoT device would reply to the ID challenge with a signature generated directly on the challenge c provided by the distributor. When verifying this lemma on the old model, the Tamarin Prover found a specific attack that would crucially break the security of the old protocol. Indeed, the attacker could act as a distributor, interface with an IoT object o_m targeted by an update release and craft the challenge c' as the concatenation of U_h and s', where s' satisfies the following properties:

- The value s' is the hash value of a value r'
- The value r' was computed as $r' := H(t' \,||\, pub^{o_m} \,||\, pub^{attacker})$
- The value t' is simply a random value

As a consequence, the attacker would obtain a signature by the IoT object defined as $sig_{o_m}^{ID'} := Sign_{prv^{o_m}}(c') = Sign_{prv^{o_m}}(U_h \| s')$, where s' has the properties above. It is easy to see how $sig_{o_m}^{ID'}$ corresponds to a proof-of-delivery signature (PoD) identical to the one that needs to be submitted by distributors to the DSC in order to obtain a cryptocurrency reward, as illustrated in Section 4.2.5. As a consequence, the attacker can effectively use this method to obtain a PoD without having to actually deliver the update, and later use this PoD to deceitfully obtain the cryptocurrency reward. This vulnerability also applies to the protocol designed by Leiba et al [9], where they have the same exploitable challenge response. After changing this step of the protocol, requiring the IoT device to generate a fresh nonce to be combined with the challenge, Tamarin could not find the same vulnerability anymore, and the security property encoded in the `PaymentOnlyIfGenerateProof` lemma was successfully verified.

Protocol Fairness for Distributors

The second security property that we formally proved is represented by the lemma in Listing 5.18. Here we are ensuring that any given distributor always receives the cryptocurrency reward if the update is successfully decrypted by the hub that requested it. More specifically, the formula defined in this lemma enforces that whenever the `UpdateReadyForIoT(IoT, U)` action fact is present in a protocol trace, then it must be that in the same trace (*i*) a certain distributor generated a zk-SNARKs proof for the same IoT device `IoT` and (*ii*) that distributor was referenced by an action fact `PaymentToD` which represents cryptocurrency payments. What is more this must be valid for all traces, which is indicated by the `all-traces` keyword.

```
lemma AlwaysPaidIfUpdateReady:
all-traces
"
  All #k IoT U
  .  UpdateReadyForIoT(IoT, U) @k

  ==> Ex #i #j pkD
      . PaymentToD(pkD, IoT) @i
      & GenProof(pkD, IoT, U) @j
"
```

Listing 5.18. *CrowdPatching* security lemma: protocol fairness for distributors

Protection Against Double Rewards

Finally, we demonstrated the impossibility for malicious distributor to deliver the update to a single IoT object and then obtain two rewards. This security property is reflected in the `MaxOnePaymentForOneIoT` lemma shown in Listing 5.19.

```
lemma MaxOnePaymentForOneIoT:
all-traces
"
  All #i #j IoT pkD1 pkD2
  . PaymentToD(pkD1, IoT) @i
  & PaymentToD(pkD2, IoT) @j

  ==>

    #i = #j & pkD1 = pkD2
"
```

Listing 5.19. *CrowdPatching* security lemma: protection against double rewards

Chapter 6

Implementation

In this chapter, we present our prototype implementation of the *CrowdPatching* protocol steps. This includes two main modules: the Ethereum module, described in Section 6.1, and the zk-SNARKs module, highlighted in Section 6.2. The former corresponds to an implementation of the three types of smart contracts employed in our protocol, namely the Super Smart Contract (SSC), the Delivery Smart Contract (DSC) and the Exchange Smart Contract (ESC). The latter consists in an implementation of the algorithms that generate and verify zk-SNARKs proofs for the specific purposes of the *CrowdPatching* protocol. Finally, in Section 6.3 we highlight two additional pieces of publicly available software, which were exploited to support the main modules and implement other features of the protocol. The complete collection of our source files, including its dependencies, can be consulted via the *CrowdPatching* repository [29].

6.1 Ethereum Smart Contracts

6.1.1 Solidity

There are several available programming languages which can be employed to develop Ethereum smart contracts [30]. Among these, we selected Solidity, more specifically employing its v0.5.16 version [31]. The Solidity programming language is the most commonly used language to build Ethereum smart contracts. It is an object-oriented high-level language influenced by C++, Python and Javascript. Furthermore, it belongs to the category of statically typed languages, meaning that the type of any variable is known at compile time as opposed to run-time.

Solidity is designed to target the Ethereum virtual machine. Indeed, Ethereum smart contracts are written in a low-level bytecode language [12]. This byte code is then executed in a a virtual machine called Ethereum Virtual Machine (EVM). As a consequence, Solidity smart contracts are translated into bytecode compatible with the EVM before they can be deployed to the Ethereum blockchain. This virtual machine can be seen as a global 256-bit computer where all Ethereum transactions are executed in synchrony. It constitutes the runtime environment for smart contracts in Ethereum. Anyone can access the EVM as long as they can run an Ethereum node.

6.1.2 Super Smart Contract (SSC)

We present here the code for the Super Smart Contract (SSC), i.e. the `SSC.sol` source file in the `ethereum/contracts` path. That is, the smart contract that needs to be deployed by a manufacturer before any new update can be released, as explained in Section 4.2.1. The implementation of the SSC presents the general structure of any Solidity smart contract, as shown in Listing 6.1. In the first line we find the `pragma solidity` instruction, which selects the specific Solidity version in use. In this case, as mentioned in Section 6.1.1, we select the 0.5.16 version. Subsequently, two external source files are imported. More specifically, we import the Solidity code for the DSC

smart contract (described in Section 6.1.3) and the ESC smart contract (described in Section 6.1.4). This is necessary because the main purpose of the SSC is to generate new DSC and ESC smart contracts, and the imported source files are used as templates to achieve this goal. Finally, all the remaining code lines are enclosed in a specific scope, i.e. inside the brackets of the `contract SSC{ ... }` construct. We continue the illustration of the SSC source file by describing the most significant elements contained in this scope.

```
pragma solidity = 0.5.16;

import "./DSC.sol";
import "./ESC.sol";

contract SSC
{
    ...
}
```

Listing 6.1. Overall structure of the SSC source file

State Variables

We declare a series of state variables, as shown in Listing 6.2 starting from line 7. They constitute the state of the smart contract, storing values on the blockchain. The first of these is the `owner` variable, which represents the address of the owner of the SSC. Storing the address of the owner is a common practice for Ethereum smart contracts, as it allows to limit the execution of certain actions to the owner only.

```
struct distributorStruct
{
    uint256 score;  // set to 0 by default
    uint lastReset; // set to 0 by default
}

address public owner;
mapping (address => bool) private childrenMap;
mapping (address => distributorStruct) private distributorsMap;
uint public distributorResetPeriod;
```

Listing 6.2. SSC state variables

The next two variables are both of type `mapping`, which is a data structure storing key-value pairs, i.e. a map. The first of these maps, called `childrenMap`, associates Boolean values to the corresponding address key. It is used by the SSC smart contract remember the address of all the DSCs and ESCs that it deploys on the blockchain. This information is fundamental to allow the SSC to receive secure commands (in the form of transactions) from its child contracts.

The second map is used to save the score of any distributor participating in the *CrowdPatching* protocol. The keys are simple addresses. It is worth noting that Ethereum addresses are directly derived from the public key of the corresponding Ethereum account. Each of these address-keys is associated with an instance of the custom struct declared at line 1. The latter contains the integer value representing the score for the distributor, as well as an additional integer value recording the last time the score was reset.

Finally, the integer variable `distributorResetPeriod` indicates the time interval after which the score of distributors is reset by the SSC. This variable is declared as `public`, as the `owner` variable is. This does not mean that anyone can modify these variables while they are stored on the blockchain. Instead, it is an indication for the Solidity compiler to create default getter functions for the corresponding variables. On the other hand, the `private` keyword employed for the remaining two variables does not make them invisible in any way. Its only effect is to

instruct the Solidity compiler to not create default getters. However, any Ethereum node can still manually access the blockchain to read those values.

Constructor

Solidity smart contracts have a constructor function, resembling many other object oriented programming languages. Indicated by the `constructor` keyword, the SSC constructor is shown in Listing 6.3. This special function is addressed by transaction that deploys the smart contract on the blockchain. The sender of such transaction is set to be the owner of the contract. This is achieved by simply assigning the address of the sender to the `owner` variable.

This constructor also accepts additional arguments, which must be attached to the transaction deploying the contract. In the context of *CrowdPatching* protocol, the creator corresponds to the manufacturer, which must decide a time interval (measured in number of weeks and days) after which score of distributors is to be reset periodically. This value is translated in an integer value and saved in the `5distributorResetWeeks` state variable.

```
constructor (uint distributorResetWeeks, uint distributorResetDays) public
{
    owner = msg.sender;
    distributorResetPeriod =
        distributorResetWeeks * 1 weeks + distributorResetDays * 1 days;
}
```

Listing 6.3. SSC constructor

Deployment of DSCs and ESCs

The main purpose of an SSC is to allow its owner, a manufacturer, to deploy new Delivery Smart Contracts (DSCs), as described in Section 4.2.2. Another is to allow any second-hand distributors (SHDs) to deploy Exchange Smart Contracts (ESCs), according to the modality illustrated in Section 4.2.3. The SSC provides this possibilities by means of two functions.

The first of these functions is called `deployDSC`, and can be found in Listing 6.4. The manufacturer can issue a new transaction addressing this function, attaching a certain amount of Ethereum cryptocurrency and the required arguments. The cryptocurrency is used for the deposit of the new DSC, while the arguments are used to initialize its state through its constructor (see Section 6.1.3). The `payable` keyword indicates that this function can indeed receive cryptocurrency.

First of all, the `deployDSC` function enforces that the sender of the transaction corresponds to the address stored in the `owner` variable. If this is not the case, the function returns without doing anything else, thus protecting itself from any malicious interaction. Indeed, Ethereum transactions are protected with signatures, so that only the real owner can identify itself as such.

The next action performed by the function is to check if the cryptocurrency sent by the manufacturer is compatible with the selected rewards. In other words, the function verifies if the deposit is enough to fund all cryptocurrency rewards, both for distributors and hubs. This is done in lines 10-12. If the cryptocurrency is not enough, the expiration time for the contract is set to 0 in line 13. In this case, the DSC deployed in the subsequent lines will not be usable by anyone, because it was created as already expired. This is done for two reasons. It avoids the possibility to create malicious contracts with unfair features for distributors and hubs. And it allows the creator to easily retrieve the deposit from a faulty DSC. Indeed, the DSC provides a function allowing its owner to withdraw funds when the contract is expired (see Section 6.1.3).

The function goes on creating a new DSC instance in line 15. This corresponds to the SSC calling the constructor of the DSC. The same arguments that were received from the function are inserted here. Additionally, the cryptocurrency value that was implicitly received is attached through the `.value(msg.value)` instruction. As a consequence, the actual digital currency is passed on to the new DSC contract. Finally, there are two other extra arguments. The address of the

sender, which is equivalent to the address of the owner. And the address of the SSC contract. The former is set as the owner of the DSC in its constructor. The latter is stored, in the DSC, as the address of its parent contract.

Lastly, the `childrenMap` entry corresponding to the address of the newly deployed DSC is flagged as `true`, and a new event is emitted. Depending on the condition checked at the beginning, the latter can be either an event announcing success, called `NewDSC`, or an event signaling a malformed DSC. In general, events are very useful in the Ethereum environment as they allow parties to listen and be notified when new events are emitted. In this case, the `NewDSC` event is especially useful as it allows distributors and hubs to discover about new update releases.

```
function deployDSC(uint _expWeeks, uint _expDays, bytes32 updateHash,
    address[] memory objectsAddresses, bytes32 pkgHash, bytes32 vkHash,
    uint256 singleRewardAmount, uint256 singleFinalRewardAmount) public payable
{
    if (msg.sender != owner)
        return;

    uint expWeeks = _expWeeks; uint expDays = _expDays;

    bool badDSC =
        (singleRewardAmount + singleFinalRewardAmount) *
        objectsAddresses.length > address(this).balance;
    if (badDSC) { expWeeks = 0; expDays = 0; }

    DSC db = (new DSC).value(msg.value)(msg.sender, address(this), expWeeks,
        expDays, updateHash, objectsAddresses, pkgHash, vkHash,
        singleRewardAmount, singleFinalRewardAmount);

    childrenMap[address(db)] = true;

    if (badDSC) emit BadDSC(address(db), msg.sender);
    else emit NewDSC(address(db));

}
```

Listing 6.4. SSC function for deploying DSCs

The second function, allowing SHDs to deploy ESCs, presents fewer instructions. Presented in Listing 6.5, the function is called `deployESC`. Here there is no need to test any condition, because anyone is allowed to to deploy an ESC. Indeed, an ESC basically consists in an offer made by a SHD to acquire the update package from a FHD, and the SHD is free to decide an arbitrary amount of cryptocurrency to offer. For an ESC to be deployed, this function only needs to receive the hash pre-image s, the expiration time and, of course, the currency to be used as an offer.

```
function deployESC(bytes32 s, uint expWeeks, uint expDays) public payable
{
    ESC eb = (new ESC).value(msg.value)(msg.sender, address(this),
        s, expWeeks, expDays);

    emit NewESC(address(eb));
}
```

Listing 6.5. SSC function for deploying ESCs

Managing the Score of Distributors

There are three functions related to the score of distributors in the SSC. One of these is called `updateDistributorPeriod`, and can be executed only by the owner. It allows the manufacturer

to change the period after which the score of any distributor is reset to 0. Another is the `getDistributorScore` function, which simply allows any Ethereum node to retrieve the score of a distributor given its address.

But the most important is the third function, called `incrementDistributorScore` and shown in Listing 6.6. As reflected in the first `if` statement, this can function can be executed only by a DSC child-contract that was previously created through the `deployDSC` function. Indeed, of the address calling the function is not flagged as true in the `childrenMap`, the function returns without executing any other instruction. Otherwise, it continues with two conditional actions:

- If the score of this distributor is not 0, and if the reset time period has passed since the last reset timestamp for this specific distributor, then the score is set to 0 and the reset timestamp is updated with the current time.
- If the last reset timestamp for this distributor is still 0, meaning that this is its first successful delivery ever, then the timestamp itself is updated with the current value.

Finally, in any case, the score is incremented. This is done regardless of any other condition because, by construction, the `incrementDistributorScore` function is called by a DSC only if the distributor has really delivered an update to an IoT device. See Section 6.1.3 for more details on how this behavior is enforced through the code of a DSC contract.

```
function incrementDistributorScore(address distributorAddress) public
{
    if (childrenMap[msg.sender] == false)
        return;

    if ( distributorsMap[distributorAddress].score != 0 &&
        ( block.timestamp - distributorsMap[distributorAddress].lastReset >
        distributorResetPeriod ) )
    {
        distributorsMap[distributorAddress].score = 0;
        distributorsMap[distributorAddress].lastReset = block.timestamp;
    }

    if ( distributorsMap[distributorAddress].lastReset == 0 )
    {
        distributorsMap[distributorAddress].lastReset = block.timestamp;
    }

    distributorsMap[distributorAddress].score++;
}
```

Listing 6.6. SSC function for incrementing the score of distributors

6.1.3 Delivery Smart Contract (DSC)

Delivery Smart Contracts (DSCs) are the most important and complex smart contracts in our system. Deployed by manufacturers, they take care of issuing cryptocurrency rewards to distributors and hubs if certain rigorous conditions are met. The source file `DSC.sol`, in the `ethereum/contracts` path, has the same overall structure illustrated for the SSC in Listing 6.1, having its constructor and its functions included inside a `contract DSC{ ... }` construct. The only instruction outside this construct is one that imports the source file of the SSC. This is necessary because the DSC needs to call a function of the SSC. On the other hand, the elements inside the construct are significantly different from the SSC, and we describe them in the following.

State Variables

The DSC presents a conspicuous number of state variables, as shown in Listing 6.7. Starting from line 9, the first three variables simply represent the addresses of the owner and the parent SSC

contract, and the expiration time associated with the DSC. Subsequently, there are two variables representing the amounts of cryptocurrency that will be sent out as rewards, one for distributors and one for hubs. Three hash values follow, corresponding to the update file, the update package and the zk-SNARKs verifying key.

The last state variable is the most complex. A map data structure called `objectsMap`, it employs addresses as key values. Its purpose is to store information about each IoT object that is targeted by the update release embodied by this DSC. In other words, this is the data structure corresponding to the list L_m described in Section 4.2.2. Each address is mapped to a special structure, declared at line 1. That is, the `struct objectStruct` containing four variables that are used throughout the rest of the contract.

```
struct objectStruct
{
    bool isMember; // set to false by default
    bytes32 r;
    bool rSet;     // set to false by default
    bool finalDelivery;
}

address public owner;          // owner's address (manufacturer's)
address public parentContract; // address of parent contract (SSC)
uint public expiration;
uint256 public singleRewardAmount;
uint256 public singleFinalRewardAmount;
bytes32 public updateHash;
bytes32 public pkgHash;
bytes32 public vkHash;
mapping (address => objectStruct) private objectsMap;
```

Listing 6.7. DSC state variables

Constructor

The constructor for the DSC is shown in Listing 6.8. In lines 6-9, the values received as arguments are simply assigned to the corresponding state variables. These values are always forwarded by the SSC, as explained in Section 4.2.1.

```
constructor (address _owner, address _parentContract, uint expWeeks, uint expDays,
    bytes32 _updateHash, address[] memory objectsAddresses, bytes32 _pkgHash,
    bytes32 _vkHash, uint256 _singleRewardAmount, uint256 _singleFinalRewardAmount)
    public payable
{
    owner = _owner;  parentContract = _parentContract;
    updateHash = _updateHash; pkgHash = _pkgHash; vkHash = _vkHash;
    singleRewardAmount = _singleRewardAmount;
    singleFinalRewardAmount = _singleFinalRewardAmount;

    if ( expWeeks + expDays == 0 )
        expiration = 0;
    else
        expiration = now + expWeeks * 1 weeks + expDays * 1 days;

    uint numObjects = objectsAddresses.length;
    for (uint i = 0; i < numObjects; i++)
        objectsMap[objectsAddresses[uint(i)]].isMember = true;
}
```

Listing 6.8. DSC constructor

Subsequently, the expiration of the DSC is set, derived from the number of weeks and days received as arguments. If these numbers are both 0, it means that the DSC is being created in an already expired state by the SSC. As a consequence, the expiration timepoint is set to 0 as well, so that the DSC is not usable and the owner can withdraw its funds right away. Otherwise, the expiration timepoint is as calculated as the sum of the current timestamp (built-in variable `now`) and the number of weeks and days converted in seconds. The result is saved in the `expiration` state variable, and essentially indicates a timepoint in the future with respect to the current time, i.e. the timestamp of the blockchain transaction that addressed this constructor.

Finally, the list of target IoT objects is processed. The list is received by the constructor in the form of an array. For each of the addresses contained in this array, a new key-value entry is inserted in the `objectsMap`. More precisely, this is achieved by setting to `true` the value of the Boolean variable `isMember`, which is part of the `objectStruct` shown in Listing 6.7.

Proof-of-Delivery (PoD) Validation

The main purpose of a DSC is to accept proof-of-delivery (PoD) submissions from distributors, in order to issue cryptocurrency payments in case the submitted values are all valid. This is achieved through the `validateDelivery` function shown in Listing 6.9. This function can be called by any Ethereum node. The necessary arguments correspond to the values listed at the beginning of Section 4.2.5. These are (*i*) the address of the target IoT object that generated the PoD signature, (*ii*) the random integer value t, (*iii*) the encryption key r, (*iv*) the hash of the encryption key s and finally (*v*) the PoD signature itself.

The `validateDelivery` function starts exactly by validating all the received values. This is done through a series of `if` statements. If any of these fail, the function returns an error and does not proceed to issue the cryptocurrency reward to the distributor. The conditions are equivalent to the ones described in Section 4.2.5:

1. The DSC must not be expired. For this condition to be satisfied, the `expiration` variable must be greater than the `now` variable, i.e. the built-in variable indicating the current time.
2. The address of the IoT object submitted by the distributor must be included in the list of targets. This is done by means of the `checkObjectExistence` function (omitted here) which simply takes an address as argument and returns a Boolean value indicating presence or absence of the address in the `objectsMap`.
3. The target IoT object must not have already received the update file. In other words, the decryption key must not have been already published for this IoT device. The Boolean value `rSet` inside the `objectStruct` has exactly this purpose. The `struct` is retrieved from the map using the address as index, and if `rSet` is true it means that the key was already revealed for this object. In this case, the contract returns.
4. The key `r` must have been obtained hashing the concatenation of the value `t`, the address of the target object and the address of the sender, i.e. the account submitting the PoD to this very function. More specifically, we employ the SHA256 algorithm, which is natively supported in Ethereum through the `sha256` function.
5. Similarly, the value `s` must have been obtained hashing the key `r`.
6. The PoD signature must indeed be a valid signature that was generated by the address of the target object, i.e. `objectAddress`. This condition is checked through the following sub-steps, corresponding to lines 19-31:
    6.1. The length of the signature is checked against a known value for Ethereum signatures
    6.2. The signature is split into three values through a portion of code written in assembly and embedded in the `splitSignature` function (omitted here)
    6.3. The message, i.e. the value upon which the signature was generated, is computed as the hash of the concatenation of the update hash and the value `s`, through the function `keccak256`. The latter applies a different hashing algorithm compared to SHA256.

6.4. The address of the signer, i.e. the address that generated this signature, can then be retrieved with the function `ecrecover`.

6.5. Finally, the condition can be actually checked: if the resulting address does not correspond to the address of the object, the contract returns an error.

```
function validateDelivery(address objectAddress, bytes16 t, bytes32 r, bytes32 s,
    bytes memory sig) public returns (string memory)
{
    if (now > expiration)
        return "DSC expired";

    if (checkObjectExistence(objectAddress) == false)
        return "Object not in the list";

    if (objectsMap[objectAddress].rSet == true)
        return "Update already delivered to this object";

    if ( r != sha256(abi.encodePacked(t, objectAddress, msg.sender)) )
        return "Invalid r: must be SHA256(t || objectAddress || msg.sender)";

    if ( s != sha256(abi.encodePacked(r)) )
        return "Invalid s: must be SHA256(r)";

    if ( sig.length != 65 )
        return "Invalid signature length";

    uint8 vSig;
    bytes32 rSig;
    bytes32 sSig;
    (vSig, rSig, sSig) = splitSignature(sig);

    bytes32 message = keccak256(abi.encodePacked(updateHash, s));
    address signerAddress = ecrecover(message, vSig, rSig, sSig);

    if (signerAddress != objectAddress)
        return "Invalid signature";

    msg.sender.transfer(singleRewardAmount);

    objectsMap[objectAddress].r = r;
    objectsMap[objectAddress].rSet = true;

    SSC ssc = SSC(parentContract);
    ssc.incrementDistributorScore(msg.sender);

    emit KeyRevealed(objectAddress, r);

    return "Delivery was successfully validated";
}
```

Listing 6.9. DSC proof-of-delivery (PoD) validation

At this point, if the contract did not return an error, it means that the overall PoD submission is indeed valid. As a consequence, the contract proceeds as follows:

1. A cryptocurrency payment is issued to the sender, i.e. the distributor, as a reward. In Solidity, this can be simply achieved through the `transfer` function, as shown in line 33. The cryptocurrency amount is indicated with the integer variable `singleRewardAmount`. The actual currency is taken from the deposit of the contract, which was funded by the manufacturer when calling the `deployDSC` function of the SSC (see Section 4.2.1).

2. Subsequently, the key `r` is stored on the blockchain, saved in the entry of `objectsMap` corresponding to the address of the target IoT device. In the same entry the value `rSet` is set to `true`, to mark the fact that the key has been revealed for this device.

3. Afterwards, the DSC communicates with its parent SSC contract. This is done to increment the score of the distributor, to account for the successful delivery at hand. The code for this action, which is not conceptually trivial, consists in two simple instructions at lines 38 and 39. The first retrieves the source code of the SSC. The second calls the `incrementDistributorScore` function providing the address of the sender as argument.

4. Finally, the DSC emits an event signaling the publication of the key for the target object. This is useful for the hub waiting for the decryption key, which is now notified.

Final-Proof-of-Delivery (PoFD) Validation

The DSC is also in charge of issuing cryptocurrency payments as reward for hubs, when they deliver a valid proof-of-final-delivery (PoFD) as described in Section 4.2.5. We omit to show the function taking care of this task in a listing. Similarly to the `validateDelivery` function presented in Listing 6.9, the following actions are performed:

- A series of conditions are enforced to make sure the PoFD submission is valid
- The cryptocurrency amount indicated by the `singleFinalRewardAmount` variable is transferred from the DSC deposit to the address of the sender
- A flag is set to `true` for the DSC to remember that this final delivery already happened

Withdrawing Funds

Another important function of the DSC is to allow the manufacturer to retrieve back the cryptocurrency from the DSC itself when it expired. This is encoded in the `withdrawFunds` shown in Listing 6.10. Two important conditions are checked. First, the DSC must indeed be expired. This can be checked through the `expiration` variable. It avoids the case of a dishonest manufacturer withdrawing the funds before time. Second, the address requesting this action, i.e. the sender of the transaction, must be the owner. If those criteria are met, the totality of the cryptocurrency balance is transferred to the sender.

```
function withdrawFunds() public
{
    if (now < expiration)
        return;

    if (msg.sender != owner)
        return;

    msg.sender.transfer(address(this).balance);

    return;
}
```

Listing 6.10. Withrdawing funds from the DSC

6.1.4 Exchange Smart Contract (ESC)

The code of an Exchange Smart Contract (ESC) is very much analogous the one of a DSC. For this reason, we avoid to show the content of the corresponding source file. It reflects the behavior described in Section 4.2.3. The corresponding `ESC.sol` source file can be found in the `ethereum/contracts` path in our repository.

6.1.5 Deployment on the Blockchain

We deployed and tested our Solidity smart contracts by means of a simulated local blockchain. This was made possible by two popular Ethereum development tools, part of the same suite of tools. One is Truffle [32], which is a development environment and testing framework for Ethereum. The other is Ganache, [33], providing a personal and local blockchain for deploying Ethereum smart contracts. For information on how we employed these tools, we refer to the Solidity Developer Manual in Appendix A.

6.2 zk-SNARKs Proving System

The goal of this section is to present our implementation of the zk-SNARKs proving system for the *CrowdPatching* protocol. In short, this proving system leverages two fundamental open-source libraries, called *libsnark* and *jsnark* and builds itself upon them. We first provide an overview of these libraries in Sections 6.2.1 and 6.2.2. Subsequently, in Section 6.2.3, we illustrate the modifications we applied to the employed libraries, and the source files we added, to obtain the specific implementation of the algorithms needed for our proposed protocol.

6.2.1 The libsnark library

To implement the zk-SNARKs system necessary for our protocol, we opted for the open-source library presented by Virza in [34] and hosted in an online repository [35]. That is, a cryptographic library called *libsnark*, written in C++ and providing an efficient implementation of zero-knowledge proof constructions. As claimed by the author, this is probably the fastest and most comprehensive suite of zero-knowledge proofs currently available.

As explained in the documentation [35], the libsnark library is a pre-processing zk-SNARK system. This means that, before proofs can be generated by a prover and verified by a verifier, one needs to select a size, circuit or system representing the NP statement that needs to be proved. Let us refer to the size, circuit or system as the *structure* of the NP statement, while the NP statement itself is just a mathematical statement as the ones we illustrated in Section 4.2.2. Once the structure of the statement is known, one can proceed running the zk-SNARKs algorithms we described in Section 2. In other words, in order to use the libsnark library to prove and verify a certain statement, one must execute the following overall steps:

1. Express the structure of the statement as one of the mathematical languages supported by libsnark. Examples of these languages are (i) the NP-complete language called Rank-1 Constraint Systems (R1CS) or (ii) the language of arithmetic circuits called Bilinear Arithmetic Circuit Satisfiability (BACS). Several others are supported.

2. Execute the zk-SNARKs *Setup* algorithm implemented in libsnark, using the statement obtained in the previous step, to create the proving key and the verifying key.

3. Execute the zk-SNARKs *Prove* algorithm implemented in libsnark, using the statement as input plus the secret and public values, to generate proofs of true statements.

4. Execute the zk-SNARKs *Verify* algorithm implemented in libsnark, using the statement as input plus the public values, to verify proofs.

Referring to the first high-level step among the ones we just highlighted, we initially decided to express our zk-SNARKs statements using the R1CS intermediate language, because it is very well supported by libsnark. Indeed, a gadget library is provided, allowing to construct R1CS instances out of modular classes. In other words, this gadget library allows to exploit existing R1CS modules and to combine them in order to obtain the target statement without having to design the full R1CS system from scratch. Many gadgets are provided, including for the SHA256 hashing algorithm or for dealing with Merkle trees.

However, libsnark does not provide any gadgets for symmetric encryption algorithms, which is crucial for the *CrowdPatching* zk-SNARKs statements illustrated in Section 4.2.2. For this reason, we turned to another library called jsnark, which is built on top of libsnark and provides an alternative way to express the structure of a statement that needs to be proved.

6.2.2 The jsnark library

The *jsnark* library [36] is an open-source project providing an alternative way of expressing the structure of an NP-statement, which can then be processed using the zk-SNARKs algorithms implemented by libsnark. In other words, it provides an alternative to the first libsnark high-level step described in Section 6.2.1. More specifically, the jsnark code consists of two main modules. The first, written in Java, is the *JsnarkCircuitBuilder*. It allows to express a statement in the form of arithmetic circuits, which is one of the languages supported by libsnark. And it does so by means of several gadgets, which can be combined to construct complex statements. Most importantly, these gadgets support more cryptographic primitives compared to libsnark. For the *CrowdPatching* protocol, we need a statement which includes symmetric encryption and hashing, and both of these are included in the jsnark gadget library.

The second module composing jsnark is an interface to the libsnark library. This is coded in C++ and allows to run the libsnark *Setup*, *Prove* and *Verify* algorithms using a statement generated with the *JsnarkCircuitBuilder* module. In other words, its purpose is to connect the two libraries together to form a complete zk-SNARKs proving system that includes all the high level steps described in Section 6.2.1.

6.2.3 Our proving system

Here we illustrate how we employed the jsnark and libsnark libraries to obtain the zk-SNARKs proving system that supports all related steps of the *CrowdPatching* protocol. The corresponding files are located in the `zksnarks` folder of our repository.

The *CrowdPatching* Circuit Generator in jsnark

The first step to code a custom zk-SNARKs proving system in jsnark is to create what is called a circuit generator. That is, a Java class that will ultimately generate the arithmetic circuit which can then be used by libsnark to prove and verify a certain statement. To create such a class, one has to extend the `CircuitGenerator` class provided by jsnark, override some of its methods and add a `main` where several methods of the `CircuitGenerator` class can be invoked. Our specific circuit generator class, located in the `zksnarks/JsnarkCircuitBuilder/src/examples/generators` path with name `CrowdPatchingCircuitGenerator.java` has the structure shown in Listing 6.11.

```java
... // Packages and imports

public class CrowdPatchingCircuitGenerator extends CircuitGenerator
{
    ... // Member variables

    // Constructor
    public CrowdPatchingCircuitGenerator() { ... }

    @Override protected void buildCircuit() { ... }
    @Override public void generateSampleInput(CircuitEvaluator evaluator) { ... }

    public static void main(String[] args) { ... }
}
```

Listing 6.11. Overall structure of the *CrowdPatching* Circuit Generator

The most significant elements of the *CrowdPatching* circuit generator are the `buildCircuit` and `generateSampleInput` methods. These are the methods we had to customize, with respect to the original `CircuitGenerator` abstract class, to obtain our statement-specific circuit implementation. As explained in the jsnark documentation [36], the purpose of the `buildCircuit` method is to is to identify the inputs of the circuit (differentiating the secret inputs from the public inputs) as well as dictating its structure by combining or connecting different gadgets. Whereas the goal of the `generateSampleInput` method is to specify the actual values of the secret and public inputs.

We start describing our `buildCircuit` method. But before doing so, it is important to remember which specific statement we are implementing. In Section 4.2.2, we introduced two different statements, almost identical expect for the size one element. The first, used by first-hand distributors to prove their possession of the update package to second-hand distributors (see Section 4.2.3), is referred to as S_E and composed as follows:

$$s = H(r) \wedge P_h = H(P) \wedge P_e = Enc(P, r)$$

The other, leveraged by distributors to prove they possess the update file to hubs (see Section 4.2.4), is indeed very similar and we refer to it as S_D:

$$s = H(r) \wedge U_h = H(U) \wedge U_e = Enc(U, r)$$

We only implemented this last statement. However, the implementation can be trivially adapted for the first statement, by changing the size of of the update file accordingly. As a matter of fact, different update releases would naturally have update files with different sizes, which means that the size of the file would have to be adjusted in any case. This is why there was no need for two implementations for these two statements.

Another crucial premise must be done. That is, to illustrate which jsnark gadgets we employed. This choice depended on the specific statement we aimed at implementing, which is S_D. According to the statement we needed support for (i) one hashing algorithm H capable of hashing files with arbitrary size and (ii) one symmetric encryption algorithm *Enc* capable of encrypting files with arbitrary size. What is more, the encryption algorithm would have had to use a key with the same size as the output of the hashing algorithm, because the protocol has been designed in a way that the key r needs to be computed as the hash of t concatenated with other values (see Section 4.2.4). All these gadgets were indeed available, provided in the form of Java classes. For the hashing algorithm, we used the `SHA256Gadget` gadget. Whereas for the encryption algorithm we used the `Speck128CipherGadget`. The Speck-128 block cipher is part of a family of lightweight block ciphers [37] designed for the Internet of Things by the National Security Agency (NSA). We selected this encryption method because it is the only one with support for the CBC mode of operation within jsnark. That is, a textbook approach for using a block cipher (capable of encrypting or decrypting blocks with fixed size only) to deal with files with arbitrary size. As shown in the following, to implement this approach we took inspiration from an existing jsnark gadget, called `SymmetricEncryptionCBCGadget`.

We divide the `buildCircuit` method in three parts. The first is shown in Listing 6.12. Here we set up the size of the variables composing the statement S_D. In jsnark, these variables have a specific type called `Wire`, which can also be used to declare arrays with the syntax `Wire[]`. We declared them as class members (see Listing 6.11, this part was omitted) but we do not initialize them right away. As a consequence, they need to be initialized here in `buildCircuit`. The first variable to be initialized in line 4 is `filePlaintextWitness8bitsWires`, an array of type `Wire[]`. This represents the variable U in the statement S_D, i.e. the plaintext of the update file. As indicated by its name, we consider each wire composing the array to have a size of 8 bits. What is more, this variable represents a secret input of the statement. Secret inputs are referred to as witnesses in jsnark, hence the presence of this word in the name of the variable. The actual initialization is applied through the `createProverWitnessWireArray` function, which must be used to initialize any secret input. As an argument, one must provide the number of wires. We compute this as the number of hexadecimal digits composing the file divided by 2. The variable `fileNumHexDigits` will be initialized in the main before the execution of the `buildCircuit` function.

Continuing with the description of Listing 6.12, subsequent variables are initialized in similar ways. Few peculiar aspects are worth noting in this context:

- For non-secret inputs, the `createInputWireArray` function is used in place of the previous one (`createProverWitnessWireArray`). An example is the variable representing the value U_h for the statement S_D, called `fileExpectedDigest32bitsWires` and initialized in line 11.
- The number of wires, i.e. the size, of each wire-array depends not only on the actual size of the represented value but also on the number of bits that each wire contains. For example, in the `rWitness8bitsWires` array each wire contains 8 bits. As a consequence, the number of wires is computed as the total number of bits (256) divided by the number of bits per wire (resulting in 32 wires). Other variables have a different number of bits per wire and thus the number of wires is calculated accordingly.
- There are two variables for the plaintext update file. This is because this file is both hashed and encrypted in the statement. As a consequence, two different inputs are needed for the different gadgets, and they also differ in the number of bits per wire. The same applies to the encryption key r, of which there are two instances.
- As mentioned before, we are using the Speck128 block cipher gadget for symmetric encryption. What is more, we are using it in combination with a CBC mode of operation to be able to encrypt arbitrary long files. As a consequence, to encrypt any file we need both a 128 bits key and a 128 bits initialization vector (IV). This is achieved by splitting the key r (256 bits) in two parts (128 bits each) that will form the Speck key and IV.

```
1  @Override protected void buildCircuit()
2  {
3      // (fileNumHexDigits is initialized in the constructor)
4      filePlaintextWitness8bitsWires =
5          createProverWitnessWireArray(fileNumHexDigits / 2);
6
7      // Key r is 256 bits: 256 / 8 = 32 bytes
8      rWitness8bitsWires = createProverWitnessWireArray(32);
9
10     // Number of 32-bits words in a SHA256 digest: 256 / 32 = 8
11     fileExpectedDigest32bitsWires = createInputWireArray(8);
12     rExpectedDigest32bitsWires    = createInputWireArray(8);
13
14     // Different wire array for the plaintext (for encryption, other is for hashing)
15     filePlaintextWitness64bitsWires =
16         createProverWitnessWireArray(fileNumHexDigits / numHexDigitsIn64bits);
17
18     // Key r (256 bits) divided into two parts (128 bits each) to obtain key and IV
19     keyWitness64bitsWires = createProverWitnessWireArray(2);
20     ivWitness64bitsWires  = createProverWitnessWireArray(2);
21
22     // (numHexDigitsIn64bits is a constant with value 64/4)
23     fileExpectedCiphertext64bitsWires =
24         createInputWireArray(fileNumHexDigits / numHexDigitsIn64bits);
25
26     ... // Second part
27
28     ... // Third part
29
30 }
```

Listing 6.12. `buildCircuit` method part one: variables initialization

The second part of the `buildCircuit` method is shown in Listing 6.13, which is exclusively dedicated to the conditions in statement S_D involving the hashing algorithm. There are expressed by the equations $s = H(r)$ and $U_h = H(U)$, and the purpose of the code in this Listing is exactly to enforce those equations. We create two instances of the `SHA256Gadget` indicating the corresponding input wire-arrays, the number of bits per wire (8) and their length. We create

two new wire-arrays to contain the result of the hashing computation, which is done through the `getOutputWires` method of the gadget. And finally we enforce the equality between the *resulting* hash values and the *expected* hash values. This is done by enforcing independent equality assertions (`addEqualityAssertion` method) to each couple of wires.

```
@Override protected void buildCircuit()
{
    ... // First part

    // Create instances of the SHA256 gadget
    SHA256Gadget fileSHA256Gadget1 = new SHA256Gadget(filePlaintextWitness8bitsWires,
        8, filePlaintextWitness8bitsWires.length, false, true, "");
    SHA256Gadget rSHA256Gadget2    = new SHA256Gadget(rWitness8bitsWires,
        8, rWitness8bitsWires.length, false, true, "");

    // Method getOutputWires() returns digest as a 32-bits wire-array
    Wire[] fileDigest32bitsWires = fileSHA256Gadget1.getOutputWires();
    Wire[] rDigest32bitsWires    = rSHA256Gadget2.getOutputWires();

    // Enforce the resulting digests and expected digests to be identical
    for (int i = 0; i < num32bitsWordsInDigest; i++)
    {
     addEqualityAssertion(fileDigest32bitsWires[i], fileExpectedDigest32bitsWires[i]);
     addEqualityAssertion(rDigest32bitsWires[i], rExpectedDigest32bitsWires[i]);
    }

    ... // Third part

}
```

Listing 6.13. `buildCircuit` method part two: hash pre-image verifications

Lastly, in the third and final part of the `buildCircuit` method shown in Listing 6.14, we enforce the symmetric encryption equality in the statement, i.e. the relation expressed with the equation $U_e = Enc(U, r)$. In particular, we employ the Speck128 block cipher gadget in CBC mode. CBC stands for cipher block chaining, and refers to a method for encrypting arbitrary long files with a block cipher, which would otherwise be capable of encrypting block-size files only. To achieve this, we adapted the code of an existing jsnark gadget, called `SymmetricEncryptionCBCGadget`.

We start this third part by declaring a new wire array that will gradually contain the result of the encryption. We apply the CBC mode of operation to this variable, which is completed at the end of the first `for` loop. At this point, we simply apply an equality assertion to each couple of wires belonging to the *expected* ciphertext and the *resulting* ciphertext, thus enforcing the equation mentioned above. As for the hashing verification, we employ the `addEqualityAssertion` provided by jsnark exactly for this purpose.

It is worth noting how, throughout the entire `buildCircuit` method, we did not assign any actual value to the involved variables. Indeed, at this stage we are only designing the structure of the statement we want to prove and verify, without having to select any valid values that satisfy the statement itself. This is why the variables are called wires: they are associated with the wires of a circuit, which can be connected and combined to obtain a certain topology regardless of the values which will flow inside them.

In the jsnark context, the actual values are assigned to the various wires in the other important method of the `CircuitGenerator` class, called `generateSampleInput`. However, assigning these values is not necessary for the party generating the zk-SNARKs proving and verifying key. This party, in the *CrowdPatching* protocol, is the manufacturer. In our jsnark code, we refer to it as the generator. Indeed, the generator only needs the structure of the statement, which can be used as input for the zk-SNARKs *Setup* algorithm explained in Section 2.3. As a consequence, the generator does not employ the `generateSampleInput` method we are about to illustrate. This difference is made explicit in the main function, which will be shown in Listing 6.16 later.

```
@Override protected void buildCircuit()
{
    ... // First part

    ... // Second part

    // Variable that will gradually become the result of the encryption
    Wire[] fileCiphertext64bitsWires = new Wire[0];

    // Implementation of the Cipher Block Chaining (CBC) mode of operation
    Wire[] expandedKey = Speck128CipherGadget.expandKey(keyWitness64bitsWires);
    Wire[] prevCipher = new Wire[2];
    prevCipher[0] = ivWitness64bitsWires[0];
    prevCipher[1] = ivWitness64bitsWires[1];
    for( int i = 0; i < filePlaintextWitness64bitsWires.length-2+1; i+= 2 )
    {
       Wire[] xored = new Wire[2];
       xored[0] = filePlaintextWitness64bitsWires[i].xorBitwise(prevCipher[0], 64);
       xored[1] = filePlaintextWitness64bitsWires[i+1].xorBitwise(prevCipher[1], 64);

       prevCipher = new Speck128CipherGadget(xored, expandedKey).getOutputWires();

       fileCiphertext64bitsWires = Util.concat(fileCiphertext64bitsWires, prevCipher);
    }

    for (int i = 0; i < fileCiphertext64bitsWires.length; i++)
    {
       addEqualityAssertion(fileCiphertext64bitsWires[i],
          fileExpectedCiphertext64bitsWires[i]);
    }
}
```

Listing 6.14. `buildCircuit` method part three: encryption verification

The `generateSampleInput` method of the `CrowdPatchingCircuitGenerator` is partially shown in Listing 6.15. In this method we repeat a very similar operation for all input and witness wires present in the `buildCircuit` method. That is, we assign valid values to all these wires, in order to prepare them for the zk-SNARKs *Prove* algorithm.

```
@Override public void generateSampleInput(CircuitEvaluator evaluator)
{
    String subStr; int wireIndex; BigInteger b; String hexString128bits;

    // Fill the wires of the file plaintext used by the SHA256 gadget
    wireIndex = 0;
    for (int i = 0; i < filePlaintextHexString.length() - numHexDigitsIn8bits+1;
         i += numHexDigitsIn8bits)
    {
        subStr = filePlaintextHexString.substring(i, i+numHexDigitsIn8bits);
        b = new BigInteger(subStr, 16);
        evaluator.setWireValue(filePlaintextWitness8bitsWires[wireIndex], b);
        wireIndex++;
    }

    ... // Fill all other wires
}
```

Listing 6.15. Main function of the *CrowdPatching* Circuit Generator

In Listing 6.15, we only show the assignment of the first wire array, which is the variable

`filePlaintextWitness8bitsWires` representing the update file in the statement. More precisely, it represents such update file for the hashing verification part only. We omit all other assignments because the operation is very much analogous for all of them. We start by initializing an integer index that is used to select a specific wire inside the array. Afterwards, we enter a `for` cycle to iterate on the `filePlaintextHexString`. This is a simple string encoding the hexadecimal characters of the update file, which must be passed by the main function to the constructor of the circuit generator as an argument. The `for` loop is constructed in a way that, in the iteration of this string, at each cycle we are advancing of a special number of characters. That is, the number of hexadecimal digits composing 8 bits, given by the `numHexDigitsIn8bits` constant which has value 2 and was initialized as a class variable. Inside the loop we do as follows: (i) we retrieve a sub-string corresponding to the two hexadecimal digits we are isolating in the current cycle; (ii) we create a new `BigInteger` Java object, which allows us to easily convert a substring with hexadecimal digits to an integer value; (iii) and we eventually set this value in the wire selected by the index-variable, which is now incremented for the next cycle.

We finally present the main function in Listing 6.16, omitting few unimportant portions. This is the function executed when the `CrowdPatchingCircuitGenerator` class is run, and coordinates the execution of various other function towards the final goal of generating the structure of the statement at hand as described at the beginning of this section.

```java
public static void main(String[] args)
{
    ... // Enforce the correct number and format for command line arguments

    boolean prover;
    if (args[0].charAt(0) == 'p') prover = true;
    else prover = false;

    // Plaintext size = 64 bytes * 4 = 256 bytes = 512 hex digits = 2048 bits
    plaintextCharString = "AJDSFAHDVKJSMN...";
    filePlaintextHexString = Util.stringToHex(plaintextCharString);
    String rHexString =
        "d4c6ecb0035d57a13e59135d29c2d4c59c26393e3032af5461f181b91e6176e4";
    String fileExpectedDigestHexString =
        "8457612244c5f5b7b2147b42ddbf859d68a78560d3f35ae4d411690cadd9a794";
    String rExpectedDigestHexString =
        "ac9e59b4e3ca66a4cb1cfb633183de3f6b6cf244b5c70da45fda3228ce71a814";
    String fileExpectedCiphertextHexString = "a52e5f3ab41c..."
    int fileNumHexDigits = filePlaintextHexString.length();

    CrowdPatchingCircuitGenerator circuitGenerator;
    if(prover)
    {
        circuitGenerator = new CrowdPatchingCircuitGenerator("my_prover",
            filePlaintextHexString, rHexString, fileExpectedDigestHexString,
            rExpectedDigestHexString, fileExpectedCiphertextHexString);
    } else
    {
        circuitGenerator =
            new CrowdPatchingCircuitGenerator("my_generator", fileNumHexDigits);
    }

    circuitGenerator.generateCircuit();
    if (prover)
    {
        circuitGenerator.evalCircuit(); circuitGenerator.prepFiles();
    } else
        circuitGenerator.writeCircuitFile();
}
```

Listing 6.16. Main function of the *CrowdPatching* Circuit Generator

More specifically, the `main` can be executed in two different modalities: as the generator of the proving and verifying keys, which is the manufacturer in our protocol; or as the prover. Let us call the generator modality *G-mode*, and the other *P-mode*. Either modality can be selected through different command line arguments, as shown in lines 5-7. As a consequence, the operations executed by the main depend on the corresponding Boolean variable.

Continuing with the content of the main function, we simply assign the secret and public values to the corresponding variables, which are hard-coded for simplicity. For example, the update plaintext file is hard-coded as a 256 bytes string in the variable `plaintextCharString`, which is then converted into hexadecimal digits. However, these variables (lines 9-18) are only needed in P-mode. Instead, the only value necessary for G-mode is the number of hexadecimal digits in the plaintext file (line 19). This difference is reflected in the following lines, where a new instance of the `CrowdPatchingCircuitGenerator` class is created. A different constructor (with different arguments) is used depending on the mode. We omit to show these two different constructors, but they simply assign the received values to the corresponding class variables.

Finally, we invoke some crucial jsnark methods. If we are in G-mode, which means we just want to create the proving and verifying key, we call the `writeCircuitFile` method. This is a method of the original `CircuitGenerator` abstract class which we did not override. What it does is to generate a description of the circuit representing the statement, and it exports it in the form of a file with extension `.arith`. This file can then be used by the jsnark-libsnark interface to generate the proving and verifying keys. Instead, if we are in P-mode, two methods are invoked, both provided by the original `CircuitGenerator` class. The first is the `evalCircuit` method, which internally calls our `generateSampleInput` shown in Listing 6.15. This is done to fill the wires with the actual values for all the elements of the statement, which is why the method is not called in G-mode. The second is the `prepFiles` method. This internally calls (i) the `writeCircuitFile`, producing the same effect as in G-mode, and (ii) the `writeInputFile` method. The latter generates a second file with `.in` extension. Its purpose is, in combination with the `.arith` file, to allow the jsnark-libsnark interface to act as the prover, which needs both the structure of the statement and the values to be assigned to all variables, secret and non-secret.

It is worth noting how there is no mention of the verifier in all the code above. This is because the verifier is implemented with the libsnark library only, i.e. without employing jsnark nor its interface with libsnark. See the next section for further details.

The zk-SNARKs Algorithms in libsnark

In the previous section we explained the jsnark code for the generator and the prover. The output of the former is a file with `.arith` extension representing the structure of the statement at hand, i.e. S_D. The latter has the same output, with the addition of a file with `.in` extension which describes the secret and public inputs to the statement. In this section, we illustrate the jsnark-libsnark interface that elaborates on those files. What is more, we are going to show the libsnark functions implementing the three zk-SNARKs entities: the generator, the prover and the verifier. The C++ source files described here are located in the `zksnarks\libsnark\libsnark \jsnark_interface\` directory path of our repository [29].

The code for the generator, which corresponds to the `generator.cpp` source file, is partially shown in Listing 6.17. First of all, this program accepts the `.arith` file generated by jsnark as in the previous section by the `CrowdPatchingCircuitGenerator` in G-mode. Afterwards, in lines 11-13, we initialize a specific libsnark variable called the *protoboard*, which is used by libsnark to encode the structure of a statement.

At this point we can create a new `CircuitReader` object, providing the `.arith` file and the protoboard as arguments. The `CircuitReader` class is the actual interface provided by jsnark, implemented in the `CircuitReader.hpp` and `CircuitReader.cpp` files. In this case, the interface is able to read the `.arith` file and to write the equivalent circuit in the protoboard, in a language that is compatible with the libsnark inner mechanisms.

We continue by obtaining the R1CS (see Section 6.2.1) constraint system from the protoboard. The constraint system is then used as an argument for the libsnark function generating the proving

key and the verifying key, now stored together in the `keypair` variable. Finally, we export these keys in the form of files in the last lines, so that they can be used by the prover and the verifier.

```
#include "CircuitReader.hpp"

...

using namespace std;

int main(int argc, char **argv)
{
    ...

    gadgetlib2::initPublicParamsFromDefaultPp();
    gadgetlib2::GadgetLibAdapter::resetVariableIndex();
    ProtoboardPtr pb = gadgetlib2::Protoboard::create(gadgetlib2::R1P);

    CircuitReader reader(argv[1], pb);

    r1cs_constraint_system<FieldT> cs = get_constraint_system_from_gadgetlib2(*pb);
    r1cs_ppzksnark_keypair<libsnark::default_r1cs_ppzksnark_pp> keypair =
        r1cs_ppzksnark_generator<libsnark::default_r1cs_ppzksnark_pp>(cs);

    ofstream pkfile;
    pkfile.open ("PK_export");
    pkfile << keypair.pk;
    pkfile.close();

    ofstream vkfile;
    vkfile.open ("VK_export");
    vkfile << keypair.vk;
    vkfile.close();

    return 0;
}
```

Listing 6.17. File `generator.cpp` implementing the *Setup* zk-SNARKs algorithm

We show the code for the prover in Listing 6.18, where we omit few non-relevant lines. Similarly to the generator, the program corresponding to this `prover.cpp` source file accepts the `.arith` file as a command line argument. In addition, the `.in` file describing the secret and public inputs must be provided as a second argument. What is more, the protoboard variable is created and initialized in the same way in lines 11-13. Afterwards, a keypair variable variable is declared in line 15. Differently from the generator, this time the key pair is not computed. Instead, in the subsequent lines until line 18, the proving key is imported from the file that was created by the generator, and stored in the corresponding component of the `keypair` object.

Subsequently, an instance of the `CircuitReader` is created. Differently from what is done in the generator, three arguments are passed: the `.arith` file, the `.in` file and the protoboard. In this way the `CircuitReader`, in addition to writing the structure of the circuit into the protoboard, it also writes the actual values for the secret inputs and the public inputs. In the context of the libsnark library, the secret inputs are referred to as the *primary* input, while the public values are referred to as *auxiliary* input. Exploiting the `CircuitReader` instance that was just created, these inputs are stored in two different variables called `primary_input` and `auxiliary_input`, in a process extending from line 22 to line 31.

Having obtained the primary input and the auxiliary input, the proof can finally be generated through the appropriate libsnark procedure in lines 33-36. More specifically, we employ the `r1cs_ppzksnark_prover`, which accepts as arguments the proving key, the secret (primary) inputs and the public inputs (auxiliary). The result, stored in the `proof` variable, is then exported into a file in the subsequent lines. In this way, it can be read by the verifier program.

```
#include "CircuitReader.hpp"

...

using namespace std;

int main(int argc, char **argv)
{
    ...

    gadgetlib2::initPublicParamsFromDefaultPp();
    gadgetlib2::GadgetLibAdapter::resetVariableIndex();
    ProtoboardPtr pb = gadgetlib2::Protoboard::create(gadgetlib2::R1P);

    r1cs_ppzksnark_keypair<libsnark::default_r1cs_ppzksnark_pp> keypair;
    ifstream in("PK_export");
    in >> keypair.pk;
    in.close();

    CircuitReader reader(argv[1], argv[2], pb);

    r1cs_constraint_system<FieldT> cs = get_constraint_system_from_gadgetlib2(*pb);
    const r1cs_variable_assignment<FieldT> full_assignment =
        get_variable_assignment_from_gadgetlib2(*pb);
    cs.primary_input_size = reader.getNumInputs() + reader.getNumOutputs();
    cs.auxiliary_input_size = full_assignment.size() - cs.num_inputs();

    const r1cs_primary_input<FieldT> primary_input( full_assignment.begin(),
        full_assignment.begin() + cs.num_inputs() );
    const r1cs_auxiliary_input<FieldT> auxiliary_input(
        full_assignment.begin() + cs.num_inputs(), full_assignment.end() );

    r1cs_example<FieldT> example(cs, primary_input, auxiliary_input);
    r1cs_ppzksnark_proof<libsnark::default_r1cs_ppzksnark_pp> proof =
        r1cs_ppzksnark_prover<libsnark::default_r1cs_ppzksnark_pp>(keypair.pk,
            example.primary_input, example.auxiliary_input);

    ofstream prooffile;
    prooffile.open("proof_export");
    prooffile << proof;
    prooffile.close();

    return 0;
}
```

Listing 6.18. File `prover.cpp` implementing the *Prove* zk-SNARKs algorithm

Lastly, we illustrate the verifier program corresponding to the `verifier.cpp` source file. Partially shown in Listing 6.19, there are two first important differences from the previous two files: the `CircuitReader.hpp` header is not included in this case and there are no command line arguments. This is because the verifier program does not use jsnark at all, nor the jsnark-libsnark interface. Instead, as explained in the following, there is only a C++ implementation, and the input values (secret and public) are hard-coded in the program. This is why the jsnark *Crowd-Patching* circuit builder only had a G-mode and a P-mode, and not a V-mode.

The first operation in this source file is to initialize the protoboard (omitted here) exactly as in previous Listings (e.g. see Listing 6.18, lines 11-13). Afterwards, the verifying key and the proof are imported from the corresponding files. Subsequently, the public inputs are assigned (hard-coded) to the appropriate string variables in lines 16-20. Indeed, the secret values are not known to the verifier, and the whole objective of this zero-knowledge proof system is that they remain unknown even when the proof is successfully validate.

```
...

using namespace std;

int main( )
{
    ...

    std::ifstream in("VK_export");
    r1cs_ppzksnark_keypair<libsnark::default_r1cs_ppzksnark_pp> keypair;
    in >> keypair.vk; in.close();
    ifstream proofFile("proof_export");
    r1cs_ppzksnark_proof<libsnark::default_r1cs_ppzksnark_pp> proof;
    proofFile >> proof; proofFile.close();

    string fileExpectedDigestHexString =
        "8457612244c5f5b7b2147b42ddbf859d68a78560d3f35ae4d411690cadd9a794";
    string rExpectedDigestHexString =
        "ac9e59b4e3ca66a4cb1cfb633183de3f6b6cf244b5c70da45fda3228ce71a814";
    string fileExpectedCiphertextHexString = "a52e5f3ab41c9...";

    int numHexDigitsInDigest = 256 / 4;
    int numHexDigitsInDigestInputVariable = 8;
    int numHexDigitsInCiphertext = fileExpectedCiphertextHexString.length();
    int numHexDigitsInCiphertextInputVariable = 16;
    const int numInputs =
        1 + (numHexDigitsInDigest / numHexDigitsInDigestInputVariable) * 2 +
            numHexDigitsInCiphertext / numHexDigitsInCiphertextInputVariable;
    int i; string subStr; string subStr1; string subStr2; int inputIndex = 1;

    VariableArray input(numInputs, "input");
    pb->val(input[0]) = readFieldElementFromHex("1");

    for (i = 0; i < numHexDigitsInDigest-numHexDigitsInDigestInputVariable+1;
        i += numHexDigitsInDigestInputVariable)
    {
        subStr =
            fileExpectedDigestHexString.substr(i, numHexDigitsInDigestInputVariable);
        char *cstr = new char[subStr.length() + 1];
        strcpy(cstr, subStr.c_str());
        pb->val(input[inputIndex]) = readFieldElementFromHex(cstr);
        delete [] cstr; inputIndex++;
    }

    ... // Loops for rExpectedDigestHexString and fileExpectedCiphertextHexString

    const r1cs_variable_assignment<FieldT> full_assignment =
        get_variable_assignment_from_gadgetlib2(*pb);
    const r1cs_primary_input<FieldT> primary_input(full_assignment.begin(),
        full_assignment.begin() + numInputs);

    const bool ans =
        r1cs_ppzksnark_verifier_strong_IC<libsnark::default_r1cs_ppzksnark_pp>(
            keypair.vk, primary_input, proof);
    printf("* The verification result is: %s\n", (ans ? "PASS" : "FAIL"));

    return 0;
}
```

Listing 6.19. File `verifier.cpp` implementing the *Verify* zk-SNARKs algorithm

However, these hard-coded values are not enough to proceed with the libsnark verifying algorithm. They need to be written inside the protoboard. To achieve this, various helper variables are computed in lines 22-29. Subsequently, three `for` loops are executed, one for each of the three hard-coded variables: one for the file expected digest, one for the expected digest for the key r and one for the file expected ciphertext. Because these `for` loops are very similar, we only show the first one and omit the other two. The key operation performed in these loops is to assign the correct values to the protoboard, through its internal `pb->val` function.

At this point we can finally verify the proof. Towards this goal, we first retrieve the primary input (non-secret values of the statement) from the protoboard in lines 47-50. The result is provided as an argument, along with the verifying key and the proof, to the libsnark function implementing the *Verify* algorithm. If the proof is valid, the return value is `true`, `false` otherwise.

Execution Time

In the previous portion of this section, we have illustrated the code for the three zk-SNARKs algorithms applied to our specific statement. In particular, the generator G and the prover P have both a jsnark-side program (written in Java) and a libsnark-side program (written in C++). Instead, the verifier has only a libsnark-side program. For instructions on how to actually execute these programs, we refer to Appendix B, where we provide a brief guide on how to run an entire zk-SNARKs example. More specifically, we outline instructions for (i) generating the proving key and the verifying key, (ii) generating a zk-SNARKs proof and (iii) verifying the same proof.

Having said that, we now provide some statistics about the execution time of each zk-SNARKs algorithm: the generator G, the prover P and the verifier V. Each algorithm is executed by a specific protocol entity, as indicated by our design in Section 4. The generator G corresponds to the manufacturer, who generates the proving key and the verifying key for a new update release. The prover P can be any distributor who is trying to convince a hub about the validity of the statement. And the hub, in turn, is acting as the verifier V. This means that none of these operations is executed by an IoT device. As a consequence, in order to measure the execution time, we always employed a non-constrained machine, i.e. a machine that does not present the same software and hardware limits that characterize an IoT device. Indeed, we employed a machine running Ubuntu 20.04, with an Intel i7 CPU and 8GB of RAM. One could argue that hubs, who act as the verifier V, can easily present more limited resources in terms of hardware and software. However, as shown by the experiments below, the verifier V requires very low computational effort, both relatively to the other algorithms and in absolute terms, which gives us high assurance about the fact that it could be executed with limited resources.

As explained at the beginning of this section, our zk-SNARKs programs apply to the following mathematical statement, where F is the secret file, r is the secret key used to encrypt it, F_e is the encrypted file and finally F_h and s are the hash digests of F and r respectively:

$$s = H(r) \wedge F_h = H(F) \wedge F_e = Enc(F, r)$$

In this statement, the only factor that could provoke different execution times is the size of its elements. Indeed, their content would not have any effect on how they are processed by the involved algorithms (the hashing function H and the symmetric encryption Enc), and the algorithms themselves are fixed. What is more, among all the elements of the statement, only three can change in size. Indeed, the output of our chosen hashing algorithm (SHA256) has a fixed length of 256 bits, which means that F_h and s have fixed size regardless of their hash pre-images. And the key r has also a fixed size of 128 bits, as it is employed by a block cipher with a specific key length (Speck128). The remaining elements, F and F_e, can vary in their size. However, F_e is the result of symmetric encryption applied to F, which means that they have the same size. In conclusion, the only parameter that can influence the execution time of the three algorithms is the size of the file F, and we obtained the consequent statistics according to this conclusion. We applied each zk-SNARKs algorithm, i.e. the generator G, the prover P and the verifier V, to four files with different sizes: (i) 0.5 kilobytes; (ii) 2 kilobytes; (iii) 5 kilobytes; (iv) 10 kilobytes. They are listed in Table 6.2.3 along with their names: F_1, F_2, F_3 and F_4. More specifically, for each algorithm we first executed its jsnark-side program, if present, and then its libsnark-side

File	Size
F_1	0.5 KB
F_2	2 KB
F_3	5 KB
F_4	10 KB

Table 6.1. Files with variable size used for the time measurements

program. We measured the execution time of both programs, using seconds as a unit, and then calculated the total with a simple addition. We employed seconds as unit because we believe it would not be meaningful to consider smaller amounts of time. For the same reason, whenever a measure goes below one second, we indicate it as “Below 1 s” for simplicity.

We show the results for the generator G in Table 6.2.3. The first observation is that the time measure is mostly influenced by the libsnark-side of the program. This was expected, and it also applies to the prover P (while the verifier V does not have a jsnark implementation). The reason is implicitly illustrated in Sections 6.2.1 and 6.2.2. That is, the fact that the jsnark library takes care of designing the structure of the statement that needs to be proved, without applying the actual zk-SNARKs algorithms, which are left to libsnark. The second and most important finding is that the total time significantly increases with the size of the file F_i.

File	File size	Time for jsnark	Time for libsnark	Total time
F_1	0.5 KB	1 s	46 s	47 s
F_2	2 KB	4 s	167 s	171 s
F_3	5 KB	11 s	440 s	451 s
F_4	10 KB	19 s	979 s	998 s

Table 6.2. Execution time measurements for the generator G

A similar set of results was obtained for the prover P, as shown in Table 6.2.3. In this case, the time measurements are slightly higher, but the pattern is analogous. Indeed, for the file F_4, the total execution time culminates with the value of 1823 seconds, i.e. 30 minutes. As a consequence, we expect these execution times to grow considerably with larger files.

File	File size	Time for jsnark	Time for libsnark	Total time
F_1	0.5 KB	1 s	50 s	51 s
F_2	2 KB	5 s	213 s	218 s
F_3	5 KB	12 s	523 s	535 s
F_4	10 KB	33 s	1790 s	1823 s

Table 6.3. Execution time measurements for the prover P

These results, relative to the generator G and the prover P, suggest that the execution of their implementation could be unfeasible with large files. To overcome this issue, we would have to apply significant optimizations to the design of our zk-SNARKs proving system, introducing complex modifications in our libsnark and jsnark implementations. However, this is out of scope for this thesis, where an optimized system was not the objective. Instead, our goal was to produce a proof-of-concept prototype, with the purpose of demonstrating that it is indeed possible to use these libraries to to design a proving system for the statement on which our protocol is based.

Finally, we show the time measurements for the verifier V in Table 6.2.3. In this case, the result present a very different pattern, where the execution time is independent of the file size.

File	File size	Time for jsnark	Time for libsnark	Total time
F_1	0.5 KB	-	Below 1 s	Below 1 s
F_2	2 KB	-	Below 1 s	Below 1 s
F_3	5 KB	-	Below 1 s	Below 1 s
F_4	10 KB	-	Below 1 s	Below 1 s

Table 6.4. Execution time measurements for the verifier V

For this reason, as we argued above, this implementation for the verifier V can be executed by devices with more limited software and hardware resources compared to the machine we employed for these experiments. This can be the case for hub (or gateway) devices, who are responsible for executing the verifier algorithm in our protocol.

6.3 Additional Software

In the previous sections we presented the implementation of two key components of our system: the Ethereum smart contracts and the zk-SNARKs proving system. Together, they allow the execution of the most important steps of the *CrowdPatching* protocol. However, few operations are not supported by these modules: (i) generating and verifying signatures offline; (ii) encrypting and decrypting files with the same Speck128 block cipher (in CBC mode) that is used by the zk-SNARKs proving system. These operations are reasonably trivial, and we were able to find existing implementations to exploit. We briefly highlight them in the following sections.

6.3.1 Digital Signatures

To generate and verify digital signatures we use the eth-crypto library [38]. Through simple Javascript programs it allows to create new identities, sign any arbitrary message and verify signatures. What is more, it is compatible with the Ethereum signature system. An example script can be found in our online repository [29] in the `signatures` folder.

6.3.2 Block Cipher in CBC Mode

For file encryption with the Speck128 cipher we use a Python implementation [39]. Indeed, this library offers a pure Python implementation of both the Simon and Speck ciphers designed by the NSA [37]. Easily installed and executed, there is also support for the CBC mode of operation through a series of primitives. Exploiting these primitives we constructed an example script which can either encrypt or decrypt a file. It can be found in our online repository [29] in the `speckcipher` folder: the script is called `my_speck_cbc.py`.

Chapter 7

Conclusions and Future Work

In this thesis, we proposed a blockchain-based decentralized protocol, allowing manufacturers to delegate the delivery of software updates to self-interested distributors in exchange for cryptocurrency payments. We introduced significant improvements with respect to the most recent research proposals in the literature, addressing key limitations with respect to the issues of scalability and practicality. We then informally analyzed the most significant threats applicable to our protocol. Furthermore, we performed a formal analysis by means of the Tamarin Prover, which provided reliant assurance on the security of the protocol, as well as on its correctness. Finally, we developed a prototype implementation, allowing to execute all steps of the protocol. This gave us further assurance about its feasibility in practice.

In our future work, we plan to refine this prototype implementation, perfecting both its inner mechanisms and its user interface. In other words, we intend to curate its performance and its usability. What is more, another possible future work is to extend our formal analysis. This can be achieved through the design of a more exhaustive model representing the protocol more accurately, as well as through the definition of new security properties to be verified.

Bibliography

[1] Statista, "Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025." https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, Webpage

[2] H. Lin and N. W. Bergmann, "Iot privacy and security challenges for smart home environments", Information, vol. 7, no. 3, 2016, DOI 10.3390/info7030044

[3] K. Kimani, V. Oduol, and K. Langat, "Cyber security challenges for iot-based smart grid networks", International Journal of Critical Infrastructure Protection, vol. 25, June 2019, pp. 36 – 49, DOI https://doi.org/10.1016/j.ijcip.2019.01.001

[4] F. J. Acosta Padilla, E. Baccelli, T. Eichinger, and K. Schleiser, "The Future of IoT Software Must be Updated", IAB Workshop on Internet of Things Software Update (IoTSU), Dublin (Ireland), June 2016

[5] J. L. Hernandez-Ramos, G. Baldini, S. N. Matheu, and A. Skarmeta, "Updating IoT devices: challenges and potential approaches", 2020 Global Internet of Things Summit (GIoTS), Dublin (Ireland), June 2020, pp. 1–5, DOI 10.1109/GIOTS49054.2020.9119514

[6] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, and R. Sirdey, "Towards Better Availability and Accountability for IoT Updates by Means of a Blockchain", 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Paris (France), April 2017, pp. 50–58, DOI 10.1109/EuroSPW.2017.50

[7] X. He, S. Alqahtani, R. Gamble, and M. Papa, "Securing Over-The-Air IoT Firmware Updates using Blockchain", Proceedings of the International Conference on Omni-Layer Intelligent Systems - COINS '19, Crete, Greece, 2019, pp. 164–171, DOI 10.1145/3312614.3312649

[8] J. Lee, "Patch Transporter: Incentivized, Decentralized Software Patch System for WSN and IoT Environments", Sensors, vol. 18, February 2018, p. 574, DOI 10.3390/s18020574

[9] O. Leiba, Y. Yitzchak, R. Bitton, A. Nadler, and A. Shabtai, "Incentivized Delivery Network of IoT Software Updates Based on Trustless Proof-of-Distribution", 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), London (UK), April 2018, pp. 29–39, DOI 10.1109/EuroSPW.2018.00011

[10] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, 2009, White Paper

[11] N. Szabo, "Formalizing and securing relationships on public networks", First Monday, vol. 2, September 1997, DOI 10.5210/fm.v2i9.548

[12] V. Buterin, "A next-generation smart contract and decentralized application platform." https://ethereum.org/en/whitepaper/, 2014, White Paper

[13] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems", SIAM Journal on Computing, vol. 18, no. 1, 1989, pp. 186–208, DOI 10.1137/0218012

[14] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps", Advances in Cryptology – EUROCRYPT 2013, Berlin, Heidelberg (Germany), 2013, pp. 626–645, DOI 10.1007/978-3-642-38348-9_37

[15] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky, "Succinct non-interactive arguments via linear interactive proofs", Theory of Cryptography, Berlin, Heidelberg (Germany), 2013, pp. 315–333, DOI 10.1007/978-3-642-36594-2_18

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture", 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA (USA), August 2014, pp. 781–796

[17] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, “Snarks for c: Verifying program executions succinctly and in zero knowledge”, Advances in Cryptology – CRYPTO 2013, Berlin, Heidelberg (Germany), 2013, pp. 90–108, DOI 10.1007/978-3-642-40084-1_6

[18] S. Ray, A. Basak, and S. Bhunia, “Patching the internet of things”, IEEE Spectrum, vol. 54, November 2017, pp. 30–35, DOI 10.1109/MSPEC.2017.8093798

[19] S. Cirani, G. Ferrari, N. Iotti, and M. Picone, “The IoT hub: a fog node for seamless management of heterogeneous connected smart objects”, 2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking - Workshops (SECON Workshops), Seattle, WA (USA), June 2015, pp. 1–6, DOI 10.1109/SECONW.2015.7328145

[20] T. T. Doan, R. Safavi-Naini, S. Li, S. Avizheh, M. V. K., and P. W. L. Fong, “Towards a Resilient Smart Home”, Proceedings of the 2018 Workshop on IoT Security and Privacy - IoT S&P ’18, Budapest (Hungary), 2018, pp. 15–21, DOI 10.1145/3229565.3229570

[21] U. Maroof, A. Shaghaghi, and S. Jha, “PLAR: Towards a Pluggable Software Architecture for Securing IoT Devices”, Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things - IoT S&P’19, London (UK), 2019, pp. 50–57, DOI 10.1145/3338507.3358619

[22] A. K. Simpson, F. Roesner, and T. Kohno, “Securing vulnerable home IoT devices with an in-hub security manager”, 2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), Kona (HI), March 2017, pp. 551–556, DOI 10.1109/PERCOMW.2017.7917622

[23] M. Ammar, G. Russello, and B. Crispo, “Internet of Things: A survey on the security of IoT frameworks”, Journal of Information Security and Applications, vol. 38, February 2018, pp. 8–27, DOI 10.1016/j.jisa.2017.11.002

[24] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A survey and comparison of peer-to-peer overlay network schemes”, IEEE Communications Surveys & Tutorials, vol. 7, no. 2, 2005, pp. 72–93, DOI 10.1109/COMST.2005.1610546

[25] The Tamarin Prover project, https://tamarin-prover.github.io/

[26] B. Schmidt, S. Meier, C. Cremers, and D. Basin, “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties”, 2012 IEEE 25th Computer Security Foundations Symposium, Cambridge, MA (USA), June 2012, pp. 78–94, DOI 10.1109/CSF.2012.25

[27] The Tamarin Prover manual, https://tamarin-prover.github.io/manual/

[28] D. Dolev and A. Yao, “On the security of public key protocols”, IEEE Transactions on Information Theory, vol. 29, March 1983, pp. 198–208, DOI 10.1109/TIT.1983.1056650

[29] CrowdPatching implementation repository, https://github.com/edoardopuggioni/crowdpatching

[30] Ethereum Online Documentation, https://ethereum.org/en/developers/docs/

[31] Solidity Online Documentation, https://solidity.readthedocs.io/en/v0.5.16/

[32] Truffle project: development environment, testing framework and asset pipeline for Ethereum, https://www.trufflesuite.com/truffle

[33] Ganache project: personal blockchain for Ethereum development, https://www.trufflesuite.com/truffle

[34] M. Virza, “On deploying succinct zero-knowledge proofs”. PhD thesis, Massachusetts Institute of Technology, 2017. DOI 1721.1/113986

[35] libsnark: a C++ library for zkSNARK proofs, https://github.com/scipr-lab/libsnark

[36] jsnark: a Java library for building circuits for preprocessing zk-SNARKs, https://github.com/akosba/jsnark

[37] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, “The simon and speck lightweight block ciphers”, Proceedings of the 52nd Annual Design Automation Conference, New York, NY (USA), 2015, DOI 10.1145/2744769.2747946

[38] eth-crypto library: Cryptographic javascript-functions for Ethereum and tutorials on how to use them together with web3js and Solidity, https://github.com/pubkey/eth-crypto

[39] Simon & Speck Block Ciphers in Python 3.x/2.x – Software library, https://github.com/inmcm/Simon_Speck_Ciphers/tree/master/Python/simonspeckciphers

Appendix A

Solidity Developer Manual

We describe the development environment we employed to compile, deploy and test our Solidity smart contracts described in Section 6.1. In particular, we opted for a simulated local blockchain environment, made possible by two popular Ethereum development tools, which are part of the same suite. The first is Truffle [32], which is a development environment and testing framework for Ethereum, allowing to easily compile Solidity smart contracts. The other is Ganache [33], providing a personal and local blockchain for deploying Ethereum smart contracts.

A.1 Installation Instructions

We provide instruction for installing our development environment on Ubuntu. More precisely, we assume that our online repository [29] was first cloned on a local directory. The next step is to open this directory in a terminal window, and to enter the `ethereum` folder. Now we can proceed installing all needed requirements:

- Install Node.js:

  ```
  curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
  sudo apt-get install -y nodejs
  ```

- Install Truffle

  ```
  npm install -g truffle
  ```

  In particular, we tested the following Truffle version:

  ```
  $ truffle version
  Truffle v5.1.48 (core: 5.1.48)
  Solidity v0.5.16 (solc-js)
  Node v12.19.0
  Web3.js v1.2.1
  ```

  To install that specific version:

  ```
  $ npm install -g truffle@v5.1.48
  ```

- Download Ganache: Instead of downloading the program from the main website [33], go to the official repository to download a specific version of the program from the assets. This implementation was tested with version 2.4.0.

A.2 Build and Deploy Smart Contracts

At this point, it possible to compile, deploy and execute the smart contracts contained in the `contracts` directory. We provide a brief guide on how to achieve this goal.

A.2.1 Run Ganache (Ethereum Local Blockchain)

We first need to run the Ganache blockchain in the background:

- Open a new terminal in the directory where the Ganache AppImage was downloaded earlier
- Execute the `chmod +x` command on the `.AppImage` file
- Execute the `.AppImage` file itself
- Select the *QUICKSTART* option

A.2.2 Configure Truffle and Ganache

Now we need to configure both Truffle and Ganache to work in coordination with each other:

- Our repository includes a Truffle configuration file in the `ethereum/truffle-config.js` path, as well as a migration configuration files (no modifications needed)
- Solidity smart contracts in this repository use compiler version 0.5.16
- As a consequence, we select the same version in the Truffle configuration file:

```
compilers: {
    solc: {
    version: "0.5.16",
    }
}
```

- Change the port used by Ganache (from the GUI) to match the one used in `ethereum/truffle-config.js` for the *development* network, which is the default network used by Truffle

A.2.3 Deploy the SSC smart contract

Finally, we deploy the SSC smart contract on the local blockchain. As per protocol design, DSCs and ESCs smart contracts can then be created by triggering the SSC to do so.

- Open a new terminal in the `ethereum` directory
- Execute the following command, which compiles the smart contracts (in this case only one, the SSC) before deploying them on the local blockchain provided by Ganache:

```
$ truffle migrate
```

A.2.4 Interact with the SCC to deploy DSCs and ESCs (and more)

We provide a Truffle script which will execute several tests, including creating new DSCs and ESCs by triggering the SSC through the proper transactions:

```
$ truffle execute truffle-executable-script.js
```

Alternatively this script can be simply modified or replaced.

Appendix B

zk-SNARKs User Manual

In this Appendix, we provide a brief guide explaining how to run an entire zk-SNARKs example by means of our implementation, illustrated in Section 6.2. More specifically, we outline instructions for (i) generating the proving key and the verifying key, (ii) generating a zk-SNARKs proof and (iii) verifying the same proof.

B.1 Prerequisites

First of all, the required software for the libsnark [35] and jsnark [36] libraries must be installed. We assume the OS to be Linux. In particular, the following was tested on Ubuntu 20.04:

- Install the packages required for libsnark through the following commands:

```
$ sudo apt-get install build-essential cmake git libgmp3-dev libprocps-dev
$ sudo apt-get install python-markdown libboost-all-dev libssl-dev
```

- Install the requirements for jsnark:
    - Install JDK8:

    ```
    $ sudo apt install openjdk-8-jdk
    ```

    - Install Junit4

    ```
    $ sudo apt-get install junit4
    ```

    - The jsnark library also requires BouncyCastle: the corresponding file `bcprov-jdk15on-159.jar` is already included in this repository; it is assumed to be placed in the `JsnarkCircuitBuilder` directory. Make it executable with the following command:

    ```
    $ chmod +x bcprov-jdk15on-159.jar
    ```

B.2 Compiling Instructions

We now explain how to compile the source files provided in our repository [29], in the `zksnarks` folder. We assume that a terminal windows was opened in the same folder, before the following instructions can be executed:

- Compile libsnark:
    - Enter the libsnark folder and create a `build` directory

```
$ cd libsnark
$ mkdir build
$ cd build
```

  - Compile all libsnark source files:

```
$ cmake .. -DWITH_PROCPS=OFF
$ make
```

- Compile jsnark:
  - Enter the circuits folder and create a new `bin` directory

```
$ cd ../../JsnarkCircuitBuilder
$ mkdir bin
```

  - Compile all jsnark source files:

```
$ javac -d bin -cp /usr/share/java/junit4.jar:bcprov-jdk15on-159.jar  $(find
    ./src/* | grep ".java$")
```

B.3 Execute the zk-SNARKs Algorithms

Finally, the instructions on how to run the three zk-SNARKs algorithms.

B.3.1 *Setup*: Generate the Proving and Verifying Keys

- Run the jsnark-side program for the generator G:

```
$ java -cp bin examples.generators.CrowdPatchingCircuitGenerator g
```

  This will produce the keys PK and VK (exported into files with hard-coded names) and the `.arith` file, which represents the arithmetic circuit for the statement at hand. The latter will be used by the libsnark-side program for the generator G.

- Indeed, a new file called `crowdpatching_generator.arith` has been created in the current directory. Now we can run the libsnark-side program for the generator G, providing the `.arith` file as a command line argument:

```
$ ~/<REPOSITORY PATH>/crowdpatching/zksnarks/libsnark/build/libsnark/
    jsnark_interface/generator crowdpatching_generator.arith
```

  This will generate the proving key PK and the verifying key VK. They are exported into files with hard-coded names, `PK_export` and `VK_export`, in the current directory.

B.3.2 *Prove*: Generate the Proof File

- Run the jsnark-side program for the prover P:

```
$ java -cp bin examples.generators.CrowdPatchingCircuitGenerator p
```

  This will create two new files. One is another `.arith` file, with a different name to distinguish it: `crowdpatching_prover.arith`. It is identical to the one created by the generator. We could avoid creating this file again and reuse the one generated by G, but we do it to represent the fact that P is independent from G, as it would happen in a real-world scenario. The other created file is a `.in` input file called `crowdpatching_prover.in`.

- Now we can run the libsnark-side program for the prover P. This will take both the `.arith` and `.in` files just created as command line arguments. Another input is the PK key file, but its name is hard-coded. Most importantly, the secret values and the public values (i.e. the non-secret values, the primary input for the NP statement) are hard-coded in the program, as explained in Section 6.2. To achieve this, we execute the following:

```
$ ~/<REPOSITORY PATH>/crowdpatching/zksnarks/libsnark/build/libsnark/
    jsnark_interface/prover crowdpatching_prover.arith crowdpatching_prover.in
```

In the last step we generated the actual zk-SNARKs proof. This is stored in a new file called `proof_export`, so that it can be retrieved by the verifier program.

B.3.3 *Verify*: Verify the Proof File

Finally, we can execute the verifier V. In this case, there is no jsnark-side program, but only the libsnark-side. This program takes no command line arguments, as the file names for the key VK and the proof are hard-coded. Most importantly, the public values (i.e. the non-secret values, the primary input for the NP statement) are hard-coded in this program as explained in Section 6.2. The following command must be executed from the terminal window:

```
$ ~/<REPOSITORY PATH>/crowdpatching/zksnarks/libsnark/build/libsnark/jsnark_interface/
    verifier
```

In this case, there result displayed as output is `PASS`, because the zk-SNARKs proof is valid. Otherwise, the result would be `FAIL`.