Thesis for The Degree of Licentiate of Engineering

Enhancing Trust in Devices and Transactions of the Internet of Things

Christos Profentzas

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2020

Enhancing Trust in Devices and Transactions of the Internet of Things

Christos Profentzas

Technical Report No 210L
ISSN 1652-876X
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

“I rebel; therefore I exist.”- A. Camus

Enhancing Trust in Devices and Transactions of the Internet of Things

Christos Profentzas
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

With the rise of the Internet of Things (IoT), billions of smart embedded devices will interact frequently. These interactions will produce billions of transactions. With IoT, users can utilize their phones, home appliances, wearables, or any other wireless embedded device to conduct transactions. For example, a smart car and a parking lot can utilize their sensors to negotiate the fees of a parking spot. The success of IoT applications highly depends on the ability of wireless embedded devices to cope with a large number of transactions. However, these devices face significant constraints in terms of memory, computation, and energy capacity.

With our work, we target the challenges of accurately recording IoT transactions from resource-constrained devices. We identify three domain-problems: a) malicious software modification, b) non-repudiation of IoT transactions, and c) inability of IoT transactions to include sensors readings and actuators. The motivation comes from two key factors. First, with Internet connectivity, IoT devices are exposed to cyber-attacks. Internet connectivity makes it possible for malicious users to find ways to connect and modify the software of a device. Second, we need to store transactions from IoT devices that are owned or operated by different stakeholders.

The thesis includes three papers. In the first paper, we perform an empirical evaluation of **Secure Boot** on embedded devices. In the second paper, we propose **IoTLogBlock**, an architecture to record off-line transactions of IoT devices. In the third paper, we propose **TinyEVM**, an architecture to execute off-chain smart contracts on IoT devices with an ability to include sensor readings and actuators as part of IoT transactions.

Keywords: Internet of Things, Distributed Systems, Blockchain, Smart Contracts, Embedded Systems, Secure Boot

Acknowledgment

First, I would like to thank Lovisa for her love and her support. I would like to thank my family and especially my mother for her support to carry on my studies.

I would like to thank my supervisors Olaf and Magnus for their support and effort to this thesis. I would like to thank the administrators Monica, Rebecca, Marianne and Eva for their help. Finally, I would like to thank my colleagues and especially Georgia, Hannah, Dimitris, Babis, Bastian, Thomas, Karl, Carlo and Beshar that make Chalmers workplace a fun and interesting environment.

Funding sources. This work is supported by the Swedish Research Council (VR) through the project "AgreeOnIT", the Swedish Civil Contingencies Agency (MSB) through the project "RIOT", and the Vinnova-funded project "KIDSAM".

Christos Profentzas,
Gothenburg, May 2020

List of Publications

Appended publications

This thesis is based on the following publications:

[A] **Christos Profentzas**, Mirac Günes, Yiannis Nikolakopoulos, Olaf Landsiedel, Magnus Almgren, "Performance of Secure Boot in Embedded Systems", *Proceedings of the 1st International Workshop on Security and Reliability of IoT Systems (SecRIoT), part of: IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS) 2019.*

[B] **Christos Profentzas**, Magnus Almgren, Olaf Landsiedel, "IoTLogBlock: Recording Off-line Transactions of Low-Power IoT Devices Using a Blockchain", *Proceedings of the 44th IEEE Conference on Local Computer Networks (LCN) 2019.*

[C] **Christos Profentzas**, Magnus Almgren, Olaf Landsiedel, "TinyEVM: Off-Chain Smart Contracts on Low-Power IoT Devices", *Accepted for publication by the 40th IEEE International Conference on Distributed Computing Systems (ICDCS) 2020.*

Contents

Abstract v

Acknowledgement vii

List of Publications ix

List of Figures xiii

List of Tables xv

1 Introduction 1
- 1.1 Motivation and Challenges . . . 2
- 1.2 Background . . . 3
  - 1.2.1 Blockchain . . . 3
  - 1.2.2 Smart Contract . . . 4
  - 1.2.3 Security Properties . . . 5
- 1.3 Domain Problems . . . 5
  - 1.3.1 Malicious Software Modification . . . 5
  - 1.3.2 Non-Repudiation of IoT Transactions . . . 6
  - 1.3.3 Sensor Readings and Actuators as part of IoT Transactions 8
- 1.4 Research Questions . . . 9
- 1.5 Scope and Delimitation . . . 10
- 1.6 Thesis Contributions . . . 10
  - 1.6.1 Device Firmware Verification (Paper A) . . . 10
  - 1.6.2 Recording IoT Transactions (Paper B) . . . 11
  - 1.6.3 Smart Contracts on Low-Power Devices (Paper C) . . . 12
- 1.7 Conclusions and Future Work . . . 12

2 Performance of Secure Boot in Embedded Systems 15
- 2.1 Introduction . . . 16
- 2.2 Background and Definitions . . . 17
- 2.3 Adversary Model . . . 19
- 2.4 Systems Overview . . . 20
  - 2.4.1 Software-based Secure Boot with U-Boot . . . 20
  - 2.4.2 Hardware Security for Secure Boot . . . 22
- 2.5 Evaluation . . . 23
  - 2.5.1 Experimental Setup . . . 23
  - 2.5.2 Evaluation Results . . . 23

2.5.2.1 Software Mechanism of U-Boot 24
2.5.2.2 TPM Hardware on BeagleBone 25
2.5.3 Discussion 26
2.6 Limitations and Discussion 26
2.7 Conclusion 27
2.8 Acknowledgments 27

3 IoTLogBlock **29**
3.1 Introduction 30
3.2 Overview and Background 32
3.2.1 Contract Signing Protocols 32
3.2.2 Smart Contracts 33
3.2.3 Blockchains and Hyperledger 33
3.3 Application Scenario and Adversary Model 34
3.3.1 Application Scenario 34
3.3.2 Adversary Model 34
3.4 System Design 35
3.4.1 Design Overview 35
3.4.1.1 Node discovery and node interaction (1*) . . . 35
3.4.1.2 Interaction between nodes and the cloud . . . 35
3.4.1.3 Verification and storage in the cloud (2*, 3*) . 35
3.4.2 Setup: Deploying new Devices 36
3.4.3 Creating and Signing Transactions 36
3.4.4 Validation with the Smart Contract 36
3.4.5 Register Transactions in the Blockchain 37
3.4.6 Security Analysis 37
3.5 Experimental Evaluation 38
3.5.1 Implementation and Experimental Setup 38
3.5.2 Evaluation of IoTLogBlock on the IoT Node 39
3.5.3 System Performance of IoTLogBlock. 42
3.6 Discussion and Limitations 42
3.7 Related Work 44
3.8 Conclusion 44
3.9 Acknowledgments 45

4 TinyEVM **47**
4.1 Introduction 48
4.2 Background 49
4.2.1 Overview of Blockchains 49
4.2.2 Smart Contracts and the Ethereum Virtual Machine . . 50
4.2.3 Payment Channels 50
4.2.4 Side-Chains 51
4.3 TinyEVM Overview 51
4.3.1 Application Scenario: Smart Parking 51
4.3.2 System Requirements 52
4.3.3 System Challenges 52
4.3.4 Threat Model 53
4.4 TinyEVM System Design 53
4.4.1 On- and Off-Chain Transactions: Three Phases 54

4.4.2 Customized Ethereum Virtual Machine 55
4.4.3 On-Chain Smart Contract 56
4.4.4 Off-Chain Smart Contract 56
4.4.5 On-Chain Commit & Challenge Period 58
4.5 Security Analysis 59
4.6 Performance Evaluation 59
4.6.1 Experimental Setup 60
4.6.2 Ethereum Virtual Machine on IoT Devices 60
4.6.2.1 Memory Requirements 60
4.6.2.2 Deployment Execution Time 62
4.6.3 Off-chain Payment Channels 63
4.6.3.1 Memory Requirements. 63
4.6.3.2 Cryptographic modules 63
4.6.3.3 Energy consumption 63
4.7 Related Work 66
4.8 Conclusion 68
4.9 Acknowledgments 68

Bibliography **69**

List of Figures

1.1 Parking Application Example 2
1.2 Blockchain Structure 3

2.1 Boot Process of Embedded Systems 17
2.2 U-Boot Configuration 20
2.3 Boot Sequence with U-Boot 21
2.4 Secure Boot with a TPM 22
2.5 The Evaluation of U-Boot 24
2.6 Evaluation of Entire Boot Process 25

3.1 Car Rental Motivating Scenario 30
3.2 Signature Exchange Protocol 32
3.3 Hyperledger Fabric Architecture 34
3.4 Electric Current Drawn from Contract Signing Protocol 40
3.5 Off-line Transction Evaluation 43

4.1 TinyEVM - Application Scenario 52
4.2 TinyEVM - System Design 54
4.3 Memory Usage of Smart Contracts 61
4.4 Deploying Time of Smart Contracts 62
4.5 Electric Current Drawn from Off-chain payments 66

List of Tables

2.1 Secure Boot Terminology 19
2.2 Secure Boot Hardware Specifications 23
2.3 Verification Overhead of TPM 26

3.1 Memory Footprint of IoTLogBlock 40
3.2 IoTLogBlock - Cryptographic Functions 41
3.3 IoTLogBlock Energy Consumption 41

4.1 TinyEVM Specifications 55
4.2 Overview of Deployed Smart Contracts 64
4.3 Memory Footprint of TinyEVM 65
4.4 Energy Consumption of TinyEVM 65
4.5 Performance of Cryptographic Operations - TinyEVM 66

Part A

Introduction

Chapter 1

Introduction

In recent years, we have experienced the rapid development of major disruptive technologies that have increased information processing and automation. One of the most prominent technologies is the Internet of Things (IoT). With the proliferation of IoT, billions of smart devices get connected and frequently interact with each other and the cloud.

This connectivity will transform major parts of our society, driven by the vast amount of data generated by connected devices. For example, a smart city may provide smart parking applications by utilizing wireless sensors on parking lots. In this example, a parking lot can interact with a vehicle to negotiate the terms of parking fees. This scenario is part of a broad spectrum of applications where smart sensors and actuators participate in billions of interactions.

In order to make these application scenarios successful, we need to investigate two key factors. First, it is inevitable for IoT devices to be exposed to security threats through Internet connections. For example, a malicious user can find ways to connect to a device and modify its software. Software modifications are especially critical in the context of IoT, as devices often control physical objects such as the cooling system of a refrigerator, or the engine of a car. Ensuring the integrity of the software is an essential property to enhance trust in IoT devices. Second, we need to store billions of interactions produced by devices that are owned or operated by different stakeholders. These interactions may be part of several agreements and transactions. With different stakeholders in action, each device needs to ensure a non-repudiation property of their transactions. It is essential for the success of IoT applications that none of the stakeholders can deny any transaction.

The remaining parts of the introduction are structured as follows. In Section 1.1, we present the challenges related to IoT architecture. In Section 1.2, we provide the necessary background on blockchains, smart contracts, and security properties in the context of IoT. In Section 1.3, we provide the domain-problems that motivated our work. In Section 1.4, we formulate the research questions that we investigate in this thesis. In Section 1.5, we present the scope of the thesis, and we formulate the research boundaries. In Section 1.6, we present the contributions of the appended papers. Finally, in Section 1.7, we conclude the introduction and present our future work.

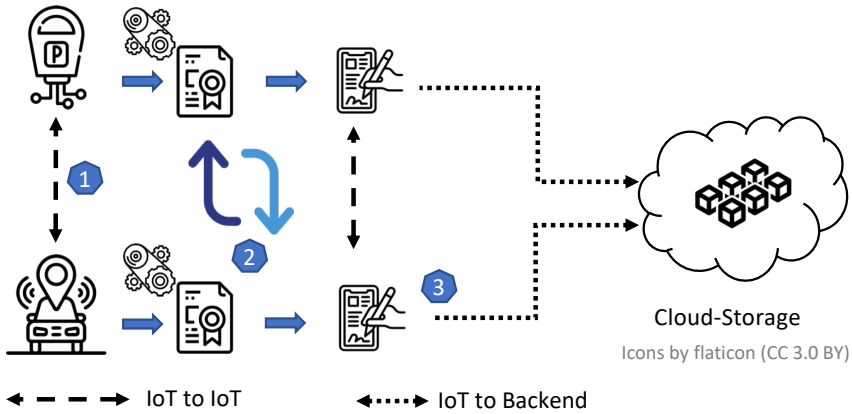


Figure 1.1: The parking application scenario. 1) The car and the parking-lot communicate via a short-range IoT protocol. 2) They reach an agreement for the parking fee. 3) Later, at any time, they can submit the final state to the cloud.

1.1 Motivation and Challenges

One useful feature of IoT systems is that cloud services may use Internet connections to have access to sensor readings and actuators in remote locations. Processing of sensor data can reveal valuable information, and often initiate control signals to actuators. In IoT systems, it is typical to use low-cost embedded devices with low-power consumption, able to operate autonomously for years. These devices experience significant constraints in terms of memory, energy, and computation capabilities. Moreover, embedded devices often have tight latency requirements with real-time properties.

As motivation, we present an example in Figure 1.1. In this example, two IoT devices interact with each other to negotiate parking fees. It is vital for IoT devices to interact with a local context (e.g., sensor data) and, at a later time, forward their interactions to the cloud where storage resources are abundant. In this scenario, IoT devices are making their interactions into two phases. In the first phase, IoT devices need to agree on the local conditions gathered from sensor data to evaluate the value of an agreement. In the second phase, IoT devices forward their interactions to the cloud, where the IoT application ensures the agreement conditions have been fulfilled. However, IoT devices do not trust each other, as they are commonly owned or operated by different stakeholders. The different interests of the stakeholders increase the risk of an inaccurate recording of IoT interactions. As these interactions may be part of several IoT agreements and transactions, it is essential to record them accurately.

The above scenario reveals a limitation of IoT devices to cope with a large number of IoT transactions. There are three key proposals to address device constraints. First, starting with the memory limitation, a way to address this challenge is to allow IoT devices to forward their transactions to the cloud, where

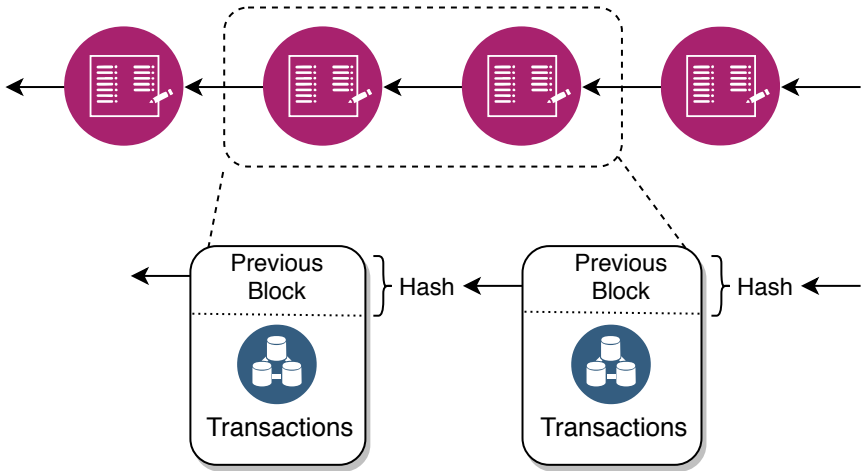


Figure 1.2: The structure of a blockchain: each block is linked with the previous block using its cryptographic hash value. Each block consists of several transactions.

hardware resources are abundant. However, cloud models face a scalability issue and fail to handle a large number of transactions. It is infeasible to send billions of transactions from IoT devices to the cloud, without exhausting the available bandwidth. Second, a way to overcome the bandwidth limitation is the proposal of edge computing. With this model, we place edge devices with enough computational resources closer to IoT devices. However, by using multiple edge devices, we create a distributed network, which requires active synchronization. It is infeasible to continuously communicate and synchronize IoT devices to edge computing without depleting the energy resources of IoT devices. Third, a way to overcome energy limitation is to allow IoT devices to perform duty cycling. However, duty cycling leads to intermittent network connections.

1.2 Background

This section provides an overview of the topics relevant to the thesis. First, we discuss the properties of distributed ledger technology (the so-called blockchain) relevant to the IoT applications and inherent challenges. Second, we introduce the concept of smart contracts and how they enforce various financial agreements. Finally, we review two major security properties in the context of resource-constrained devices.

1.2.1 Blockchain

A blockchain is essentially a type of distributed ledger [1], where multiple nodes keep replicated blocks of transactions as an append-only data structure. The data structure is a linked list that chains together the next block by using the cryptographic hash value of the previous block. The structure of a

blockchain is visualized in Figure 1.2. With this structure, it is trivial for each node in the network to verify the validity of every new appended block. Any change to an older block inherently invalidates all chained blocks. As blocks are replicated over multiple nodes in the network, the nodes need to reach a consensus regarding the order of the blocks. Several algorithms have been proposed to reach a consensus in a blockchain, such as Proof of Work (PoW) [2], Proof of Stake (PoS) [3], and Practical Byzantine Fault Tolerance (PBFT) [4].

For example, Bitcoin [2] uses a PoW algorithm. In this algorithm, every node can participate in the consensus algorithm. A node in the network may propose the next block by solving a probabilistic mathematical problem. When a node solves the mathematical problem, it creates and broadcasts a new block to the network for validation.

The increasing deployment of large-scale blockchains reveals several scalability issues, for example, low transaction throughput [5–7]. The append-only data structure creates an ever-growing blockchain which reaches large sizes in a short period of time [8]. The number of transactions that the network can handle is relatively small compared to traditional transaction systems. Currently, the throughput is an order of tenth per second [5], compared to an order of thousands when using traditional financial systems, such as credit cards. The research community has proposed several ways to scale the blockchains, such as sharding [9, 10], consensus algorithm variations [1], trusted execution [11], side-chains [12], payment channels [13, 14], and payment networks [15, 16].

In the context of IoT, we can identify three resource-intensive requirements that make blockchains challenging to adopt in resource-constrained devices. First, the consensus algorithms require extensive processing power from participating nodes. Second, participating nodes require sufficient network bandwidth to broadcast each block. Third, the ever-growing blockchain requires sufficient memory to store the complete data structure.

1.2.2 Smart Contract

A smart contract is an executable program replicated in a blockchain. The primary use for a smart contract is the digital representation of financial agreements between untrusted parties. The parties assign a public key to a smart contract that anyone can use and send transactions to it through a blockchain network. Smart contracts have variables to store their states in a blockchain, and the execution of it may update the state of a blockchain. Each node participating in the consensus process of a blockchain executes and validates every smart contract. It is typical to use a Virtual Machine (VM) and write a smart contract in a high-level programming language [17].

The current design of VMs for smart contracts assumes a sequential execution with no support for parallel or concurrent execution. Moreover, smart contracts do not have access to data outside of a blockchain network, limiting their ability to handle events of the real world. With the current design, in order to include sensor data inside a smart contract, we need to use a third party, the so-called Oracles [18]. We argue that Oracles, acting as third parties, oppose the fundamental principle of blockchain to replace central authorities.

In the context of IoT, we can identify two main obstacles to adopt smart contracts on IoT devices. First, smart contracts are written in high-level

languages which are commonly too memory-intensive for resource-constrained devices. Second, the virtual machines of current blockchains do not provide any opcode to handle sensor data or send signal to actuators.

1.2.3 Security Properties

The security of payment protocols is crucial for the success of many applications, especially in IoT. In this thesis, we focus on two security properties of IoT devices. First, we focus on the **integrity** [19] of the software running on the embedded device. Second, we focus on the **non-repudiation** [19] property of transactions between two or multiple untrusted parties, commonly found in our application scenarios.

It is common for manufacturers of embedded devices to divide the boot process into several steps due to memory constraints. Commonly, they divide the boot process of embedded devices into three steps. First, the boot-ROM is the initial software running when a device starts. Second, the boot-loader is the software responsible for transferring control to the operating system. Third, the operating system is the final software that starts the application(s) of the device. We investigate **Secure Boot** [20], as a way to ensure the integrity of software of all the above steps.

One way to achieve **non-repudiation** is to exchange cryptographic signatures of a pre-agreed message such that either both parties or none obtain a signed message. We use the term **fair exchange** to refer to the property of signature exchange protocols where either both or none of the participants obtain the cryptographic signature. A **contract signing protocol** allows two or multiple parties to exchange signatures in a fair-way. We group the contract signing protocols in three categories: protocols with an online Trusted Third Party (TTP) [21], protocols without TTP (e.g., time commitments [22]), and protocols with an off-line TTP (e.g., optimistic [23]).

1.3 Domain Problems

In this section, we introduce three domain-problems that motivate our work: a) malicious software modification, b) non-repudiation of IoT transactions, and c) inability of IoT transactions to include sensors readings and actuators.

1.3.1 Malicious Software Modification

In the first problem-domain, we encounter malicious modifications of the software running on IoT devices. The possibility of these modifications decrease the trust in the data produced by the sensors and actuators commonly found in an IoT application.

Motivation. As the Internet of Things (IoT) brings connectivity to everyday objects, embedded devices get exposed to numerous cyber-attacks [24]. Device manufacturers commonly design their systems to work in isolation with little exposure to Internet connections. With Internet connectivity, internal structures, such as the firmware of the device, are now exposed to cyber-attacks such as malicious software modifications. These modifications can remain

unnoticed for an extended period of time, which raises questions regarding the trust in the data originating from the device.

For example, a possible malicious user can find ways to remotely connect into an IoT device and replace the existing firmware with one containing malicious code. The malicious code will be the first function to be executed every time the device boots-up, prior to any security protection mechanism. Therefore, any malicious activity of the device can remain hidden.

A common approach to detect software modifications is to ensure the integrity of the software during the boot process, the so-called *Secure Boot*. It is common to find *Secure Boot* inside a desktop computer [25]. However, in the domain of embedded systems, it is quite rare.

Challenges. The process of *Secure Boot* in any device requires the extension of the system with additional software or hardware. The resource-constrained devices need to overcome two significant overheads to adopt these additional components. First, it requires an extension of the boot-up software or hardware. This additional software or hardware increases the complexity of the boot-up process, and it depletes the limited resources of the embedded device. Second, it introduces extra sequential steps in the boot-up process, which delays the start-up time and may affect the usefulness of the user-application.

When it comes to embedded devices, we can find a broad range of devices. This variety makes it hard to adopt any *Secure Boot* process uniformly. Moreover, IoT applications have different delay and response requirements. One main challenge is to recognize and adopt the overheads introduced by *Secure Boot*.

Related work. There is a large body of research in the field of *Secure Boot*. Khalid et al. [26] discuss the difference between Secure and Trusted Boot, and they further evaluate the performance overhead on FPGA boards. Lebedev et al. [27] give examples of a remotely attested system using FPGA boards.

From the practitioner's side, Google's Chromium OS uses Verified Boot to build a chain of trust [28]. Asokan et al. [29] focus on solutions regarding firmware updates on large-scale IoT deployments. For resource-constrained devices, Boot-IoT [30] proposes an authentication scheme for secure bootstrapping. We can see the need for *Secure Boot* on different device sizes ranging from consumer-level printers to industrial robots [31, 32].

Open problems Even though researchers propose several methods to ensure the integrity of device firmware, most of them have not been evaluated on resource-constrained devices. Moreover, there are millions of connected devices without any measure to verify the software they run. In most cases, there is no way to update or change the software after the deployment of IoT devices.

1.3.2 Non-Repudiation of IoT Transactions

In the second domain-problem, we focus on the challenges involved in ensuring non-repudiation of IoT transactions. In our case, the non-repudiation property would be violated if a completed IoT transaction is wrongly recorded or not stored at all, or if a misbehaving node is not immediately reported.

Motivation. In a typical IoT application, small devices need to interact frequently, producing billions of transactions that their stakeholders want to

store. In this case, the interacting IoT devices commonly do not trust each other, as they are, for example, owned or operated by different stakeholders. Moreover, the distributed nature of IoT applications makes it challenging to record such interactions coming from distrusting nodes.

Blockchain technology, as described in Section 1.2, stores blocks of transactions in the form of an immutable linked list. Moreover, blockchain promotes a verifiable property of transactions in a network of untrusted parties. This property makes the blockchain an ideal candidate for storing the billions of transactions produced by IoT devices.

As motivation, we present an application in Figure 1.1, where one device provides a service to another for an exchange of money. The small embedded devices are limited in terms of the connectivity that they can establish. Thus, it becomes infeasible to verify each transaction by contacting an online verification service every time. It is more reasonable for the devices to interact off-line using the online services only in case of a dispute. The devices can ensure a non-repudiation property of their transactions by applying a contract signing protocol to sign their transaction in a fair-way [33]. In this thesis, we are motivated by the advances in the cryptographic capabilities of embedded systems [34], and we identify new opportunities for applying cryptographic protocols to sign IoT transactions.

Challenges. In order to operate a blockchain network with IoT nodes, we face three key challenges. First, blockchain nodes require sufficient network bandwidth to broadcast a large number of messages. Blockchains require network bandwidths that is beyond the capacity of resource-constrained devices. Even in the case of a broadband connection (e.g., 5G), its energy requirements will deplete the battery of the device too quickly. Second, it is inevitable for devices to off-load their transactions to the cloud as their storage capacity is insufficient to store the complete blockchain. Third, nodes participating in a blockchain network need to use heavy-cryptographic operations to verify each block.

The resource limitations of IoT devices lead to three relaxations. First, nodes need to operate, assuming intermittent network access. They need to utilize edge connections to reach cloud-based blockchains. Second, nodes need to act optimistically and create off-line transactions. In case of a conflict, they should contact cloud services to resolve it. Third, nodes need to use lightweight protocols to address device limitations in terms of memory, computation, and energy consumption.

Related work. Recent studies on the affordability of cryptographic operations on embedded devices [35] [36] inspire the idea of utilizing a blockchain in the context of IoT. However, current architectures focus on power-full devices, and they overlook the limitations of the resource-constrained devices [37]. Several architectures [38] [39] identify the ability of blockchains to provide data provenance. While other studies, such as AGasP [40] and Edgechain [41], identify different applications of blockchains within IoT.

Open problems. Even though state-of-the-art has increased the adoption of IoT devices within blockchains, there are still significant limitations. We identify three key limitations of the existing approaches. First, they are not able to ensure non-repudiation of IoT transactions. Non-repudiation is essential, as stakeholders often have a conflict of interest, and we need to ensure they

cannot deny any transaction. Second, current architectures require significant communication and fixed connection points, which overlooks the limitation of resource-constrained devices. For example, in prominent designs [41] [42], nodes participating in the networks need to actively communicate and broadcast their transaction, demanding considerable energy capacity. Third, data tampering and device impersonation attacks [43] raise concerns about the secure collection of sensor data.

1.3.3 Sensor Readings and Actuators as part of IoT Transactions

In the third domain-problem, we encounter the inability of IoT transactions to include sensors readings and actuators.

Motivation. We can imagine a smart car and a parking lot negotiating hourly parking fees by utilizing sensor data (see Figure 1.1). After the smart car and parking lot reach an agreement, they can publish it on the blockchain. The involved parties can utilize payment channels (see Section 1.2) to perform off-line transactions, and at a later time, enforce the payments by publishing a final state in the blockchain.

The parking application reveals two main challenges. First, the current technology of blockchains and smart contracts assumes powerful nodes. For example, Ethereum [17] uses a virtual machine to execute smart contracts, and nodes need to connect to the blockchain both to upload their frequent transactions and to query updates. Second, smart contracts do not have access to sensor readings and actuators. Thus, an application using a smart contract is unable to utilize the local context (sensor data) of the environment (e.g., parking slot).

We recognize the following two properties for the above application. First, the application needs to utilize sensor readings and actuators inside a smart contract. This ability will allow the application to make decisions based on local conditions. Second, the application expects the exchange of transactions to happen quickly. In our application, we expect a car and a parking lot to finish a transaction in the order of seconds.

Related work. The increasingly large-scale deployments of blockchains reveal the limited scalability of its payment system [6, 7]. The scalability of blockchains is an active field with several proposals, such as sharding [9, 10], consensus algorithm variations [1], and trusted execution [11]. In this thesis, we look into three prominent proposals for scaling the payment system of blockchains. First, research studies propose the extension of the blockchain system with payment channels [13, 14]. With a payment channel, two or more parties can swiftly exchange off-line payments, and at a later time, update the blockchain with the final state. Second, other research studies extend the idea of payment channels with payment networks [15, 44, 45]. With a payment network, nodes can reuse existing payment channels to route payments without the need to reopen and close new channels. Third, the proposal of side-chains [12, 46] attempts to solve the scalability issues in a different way. The idea of side-chains is to create and maintain one or multiple separate off-line chains connected to the main blockchain. A published smart contract on the main blockchain acts as a root of a side-chain. Finally, an exit function allows off-chain nodes to

claim tokens from the main blockchain.

Another research area focuses on the inability to include data from the physical world in blockchains. Several proposals focus on third parties that act as an Oracle [18] inside the blockchain. For example, TownCrier [11] provides HTTPS-enabled data from websites. Moudoud et al. [18] propose an Oracle for industrial supply chains.

Open problems. Even though payment channels offer a solution to scalability issues of blockchains, there are two main challenges to make payment channels applicable in the context of IoT. First, a payment channel requires the exchange of several messages. These requirements are questionable on resource-constrained devices. Second, the off-line exchange of payments uses heavy-cryptographic computations to sign each state. There is a lack of empirical analysis of the feasibility of IoT devices to execute such off-line payments.

Another issue with existing proposals is that they do not integrate any local context (e.g., sensor data) in IoT applications. Smart contracts are not well designed to handle input from the outside world. The current data providers (e.g., Oracles) do not give a direct way for a smart contract to include sensor readings and actuators as part of a transaction. Overall, we recognize a gap between high-level blockchain architectures and the need to integrate sensor data and actuators of IoT devices.

1.4 Research Questions

Based on our discussion and the challenges presented in Section 1.3, we introduce three main research questions that motivate this thesis:

- **RQ1:** How can we ensure the software integrity of IoT devices?
- **RQ2:** How can we provide non-repudiation for IoT transactions?
- **RQ3:** How can sensor readings and actuators become part of IoT transactions?

RQ1 becomes particularly important when considering the untrusted nature of the IoT environment. Any malicious user can try to modify the device software through Internet connections. There are several approaches to ensure the integrity of the software when it comes to desktop computers. However, resource-constrained devices face different performance and run-time overheads compared to desktop computers.

RQ2 and RQ3 become important when considering billions of IoT transactions coming from small embedded devices. The ability to store billions of transactions is quite challenging when applied to resource-constrained devices. We need to ensure that none of the stakeholders can deny any transaction between IoT devices. On the other hand, IoT applications need to include sensor readings and actuators as part of IoT transactions.

1.5 Scope and Delimitation

This thesis is a systems research study that covers aspects of data verification, recording, and processing in the context of the Internet of Things (IoT). The thesis uses empirical experimentation on a system level to investigate properties relevant to the deployment of IoT devices. Our motivation stems from recent advances in cryptographic operations on IoT devices and the potential use of distributed ledgers to store IoT transactions. This thesis addresses challenges related to the integration of resource-intensive technologies on resource-constrained devices.

The integrity of the software running on the devices is an important security property when it comes to the massive adoption of IoT devices. These connected devices are vulnerable to cyber-attacks and penetration from malicious users. Any intruder can modify the executable files of the device through Internet connections and compromise essential functions of the system. The prevention of malicious software modification is critically important in the context of the IoT, as it is common for these devices to include sensors and actuators in safety-critical environments, e.g., the fire-alarm of a building.

Recording of interactions between IoT devices, as in any other distributed system, is an essential property of the system. Multiple parties are involved in the process, and they do not trust each other as they may face a conflict of interest. However, maintaining a distributed ledger of interactions is challenging for resource-constrained devices. Distributed ledgers are resource-intensive technologies, and small embedded devices face constraints in terms of memory, computation, and communication capacity.

Finally, we limit our study within two boundaries. First, our study investigates techniques to ensure the integrity of device software limited to detect malicious firmware modifications. Our thesis does not introduce a general protection mechanism against cyber-attacks. Second, the privacy of IoT transactions is limited to existing techniques suitable for blockchains. Our study does not propose any design alteration on the existing blockchains.

1.6 Thesis Contributions

In this section, we outline the contributions of the appended papers in relation to our research questions. The papers address the domain-problems mentioned in Section 1.3.

1.6.1 Device Firmware Verification (Paper A)

In Paper A, we identify the performance trade-offs of performing **Secure Boot** on medium-scale embedded systems. This work builds on the observation that modern embedded devices have increased capabilities of performing cryptographic operations. As a result, conventional techniques designed initially for desktop computers are becoming applicable to resource-constrained devices. With that in mind, we target medium-scale embedded devices, and we show how different cryptographic algorithms affect the boot-up time. In this way, we contribute towards **RQ 1** and identify trade-offs when it comes to **Secure Boot** with relation to time performance.

We follow a two-step approach. First, we identify two secure boot techniques that are applicable on medium-scale embedded devices: i) a software-based technique where the verification is part of the boot-loader, and ii) a hardware-based technique where we extend the device with additional hardware to store the verification software in a trusted way. Second, we analyze the trade-off of secure boot techniques concerning time performance. We emphasize time performance as we identify it as a crucial element for several applications. For example, safety-critical embedded devices usually have strict boot-up times as any delay can lead to life-risks. Another example is a home appliance, where users frequently restart their devices, and they expect devices to become quickly available. In that case, the boot-time can determine the usability of a device.

For our experiments, we use wide-spread platforms such as Beaglebone and Raspberry Pi. We implement and evaluate two secure boot techniques on these platforms. We investigate the trade-offs of a software-based and a hardware-based technique. In the case of the software-based technique, we show a time overhead of 4% relative to the original boot process. In the case of the hardware-based technique, the additional overhead is 36%. This relatively high overhead of the hardware-based technique is reasonable as the extra hardware is used for additional security purposes rather than increasing performance.

1.6.2 Recording IoT Transactions (Paper B)

In Paper B, we introduce **IoTLogBlock**, a novel architecture for recording IoT transactions using a blockchain. With this architecture, we address the challenges of storing records of interactions coming from IoT devices that do not necessarily trust each other. The devices are owned and operated by different entities with a conflict of interest. We investigate the challenges of managing and registering untrusted IoT devices using a blockchain network. Through this work, we contribute towards **RQ 2** by designing and implementing an architecture to record IoT transactions. We show the trade-offs of using a blockchain to record transactions coming from resource-constrained devices.

We motivate our work using a car rental service application. In this application, a company provides rental cars distributed throughout a city. A customer willing to rent a car needs to reach an agreement with the rental company. In this case, the stakeholders are known in advance, but they are facing a conflict of interest, and they do not necessarily trust each other.

IoTLogBlock focuses on resource-constrained devices, and it consists of three building blocks: a) a lightweight contract signing protocol, b) a smart contract, and c) a blockchain. The contract signing protocol achieves non-repudiation of off-line transactions by utilizing an on-line entity to report any misbehavior. We design a smart contract to act as the on-line entity to report complete transactions or misbehaving nodes. The blockchain network is used to store the transactions permanently.

We took into consideration three potential threats when designing our architecture. First, an adversary may try to skip reporting transactions. Second, we consider an adversary that may try to remove stored transactions. Third, an adversary can try to include unverified transactions. In our design, we took care of the above threats and motivate our prevention measures.

We evaluate our design on recourse-constrained devices like TI CC2538 [47]

and quantify the performance of **IoTLogBlock** in terms of memory, computation, and energy consumption. TI CC2538 has a 32-bit ARM Cortex M3 CPU at 32 MHz, 32 KB of RAM, 512 KB of ROM, and 802.15 radio transceiver. Our results show that a recourse-constrained device can create and sign a transaction within three seconds on average. Finally, we evaluate the devices using different network scenarios with edge connections ranging from ten seconds to over two hours.

1.6.3 Smart Contracts on Low-Power Devices (Paper C)

In Paper C, we introduce **TinyEVM**, a novel system to generate and execute off-chain smart contracts. With **TinyEVM**, smart contracts have access to sensor readings and actuators of IoT devices. **TinyEVM** allows IoT devices to create and perform off-chain payment channels considering the resource-constraints of IoT devices. Through this work, we contribute towards **RQ 2** by showing the trade-offs of executing off-chain payment channels on resource-constrained devices. We are considering a broad spectrum of applications where the myriads of IoT devices are frequently exchanging payments in two steps. First, they use their sensor data and actuators upon their environment to agree on payment conditions. Second, they enforce these payments by publishing a final state.

We motivate this work using a parking service scenario. Through this scenario, we show the challenges of negotiating a parking place between two parties using a smart contract. In order to make this scenario feasible, we need to ensure two properties. First, we need the utilization of the sensor data to determine the value of the parking place. Second, we need to enable the parties to finish their interactions and complete a transaction in a matter of seconds.

TinyEVM focuses on resource-constrained devices like TI CC2538 [47], and it consists of three components: a) publishing the on-chain smart contract, b) creating off-chain channels, and c) committing on the on-chain. We achieve these steps by separating the logic of on- and off-chain transactions by designing two smart contracts. Moreover, we design, implement, and extend with IoT opcodes an Ethereum Virtual Machine to execute smart contracts on resource-constrained devices.

We investigate the trade-offs to execute smart contracts on recourse-constrained IoT devices. We test our system with 7,000 publicly verified smart contracts, where **TinyEVM** manages to deploy 93% of them. Finally, we evaluate the execution of off-chain smart contracts in terms of run-time performance, energy, and memory requirements. Notably, we find that recourse-constrained devices can deploy a smart contract in 215 ms on average, and they can complete an off-chain payment in 584 ms on average.

1.7 Conclusions and Future Work

This thesis is motivated by the emergence of IoT technologies and their impact on our society. With IoT, billions of devices get connected to the Internet and interact with each other. IoT devices can potentially create billions of transactions and enable new application scenarios. With this work, we target

challenges related to accurately recording IoT transactions from resource-constrained devices. We identify three domain-problems: a) malicious software modification, b) non-repudiation of IoT transactions, and c) inability of IoT transactions to include sensors readings and actuators.

For the first domain-problem, we raise the question: How can we ensure the software integrity of IoT devices? We investigate this question in Paper A. In this paper, we perform an empirical evaluation of **Secure Boot** on embedded devices. For the second domain-problem, we raise the question: How can we provide non-repudiation for IoT transactions? We investigate this question in Paper B. In this paper, we propose **IoTLogBlock**, an architecture to recording off-line transactions of IoT devices. For the third domain-problem, we raise the question: How can sensor readings and actuators become part of IoT transactions? We investigate this question with Paper C. In this paper, we propose **TinyEVM**, an architecture that includes sensor readings and actuators by executing off-chain smart contracts on recourse-constrained devices.

Based on insights from our work in the integration of IoT devices with blockchains, we identify two interesting topics for future work. First, there is the open topic of the privacy of stored transactions in a cloud-based blockchain. Second, the off-chain payment channel can be extended into payment networks and improve the scalability of IoT transactions.

Part B

Papers

Paper A

Chistos Profentzas, Mirac Günes, Yiannis Nikolakopoulos, Olaf Landsiedel, Magnus Almgren,

Performance of Secure Boot in Embedded Systems

Proceedings of the 1st International Workshop on Security and Reliability of IoT Systems (SecRIoT), part of: IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS) 2019.

Chapter 2

Performance of Secure Boot in Embedded Systems

With the proliferation of the Internet of Things (IoT), the need to prioritize the overall system security is more imperative than ever. The IoT will profoundly change the established usage patterns of embedded systems, where devices traditionally operate in relative isolation. Internet connectivity brought by the IoT exposes such previously isolated internal device structures to cyber-attacks through the Internet, which opens new attack vectors and vulnerabilities. For example, a malicious user can modify the firmware or operating system by using a remote connection, aiming to deactivate standard defenses against malware. The criticality of applications, for example, in the Industrial IoT (IIoT) further underlines the need to ensure the integrity of the embedded software.

One common approach to ensure system integrity is to verify the operating system and application software during the boot process. However, safety-critical IoT devices have constrained boot-up times, and home IoT devices should become available quickly after being turned on. Therefore, the boot-time can affect the usability of a device. This paper analyses performance trade-offs of secure boot for medium-scale embedded systems, such as Beaglebone and Raspberry Pi. We evaluate two secure boot techniques, one is only software-based, and the second is supported by a hardware-based cryptographic storage unit. For the software-based method, we show that secure boot merely increases the overall boot time by 4%. Moreover, the additional cryptographic hardware storage increases the boot-up time by 36%.

2.1 Introduction

The Internet of Things (IoT) will bring connectivity to everyday objects and devices, including vehicles [48], autonomous robots [49], and smart home appliances [50] (e.g., smart vacuum cleaners, smart cookers, smart heaters). While Internet connectivity allows numerous new applications and use-cases, it exposes devices to the security threats of the Internet: if we connect a device to the Internet, it will certainly be attacked and potentially penetrated, i.e., intruders can read data or even modify executable system files. Such modifications are especially critical in the context of the IoT, as the devices often control physical objects such as the cooling system of a refrigerator or the engine of a car.

Today, we commonly find a secure boot process in regular computer systems, including personal computers [25], data centers [51] and also portable devices such as smartphones or tablets [52]. Those computers usually include extra hardware (e.g., a Trusted Platform Module) to ensure the integrity of the firmware and the operating system during the boot process. However, in the domain of embedded systems, secure boot is often overlooked. Therefore, common IoT devices rarely secure the boot process and fail to assure software free from manipulation [53]. The absence of secure boot opens the door to attacks on mission-critical IoT systems. For instance, recent work demonstrates attacks that alter the firmware of interconnected industrial robots [32]. A secure boot mechanism would have detected such modifications during the boot-up process.

Securing the boot process in embedded devices leads to two significant overheads. Firstly, a secure boot process adds additional hardware and complexity. Embedded devices need efficient and simple designs since they face several constraints when it comes to energy consumption and memory capacity. Secondly, verifying the integrity of the operating systems or firmware adds a further delay to the boot process. In some applications, longer boot time may not be affordable. For example, micro-controllers used in the automotive industry should be able to boot almost immediately, preferably in the sub-second domain, so that the vehicle can be used directly after ignition [54]. Other examples are smart home devices where users frequently turn them on and off, like vacuum cleaners and electric kettles. For those devices, longer boot-up times lead to less usability for their users.

Embedded devices include a range of different microcontrollers, which we can classify into three groups: small scale (8–16 bit), medium scale (16–32 bit), and sophisticated micro-controllers. This paper focuses on securing the boot process of medium-scale microcontrollers (e.g., Raspberry Pi, Beaglebone) equipped with an embedded operating system.

The paper makes two contributions: (1) We examine two different approaches to secure the boot process of an embedded device. (2) We show the performance and runtime overhead of the secure boot for those two approaches. Our results underline the trade-offs between security and performance: we present the tradeoffs ranging from 58ms to 245ms for a software-based secure boot, which is 5.6–16% of the boot-loader execution time, and 1.4–4% of the entire boot time (4065ms) of an off-the-shelf Linux distribution. For the hardware-based secure boot technique, we observe an overhead of 1900ms,

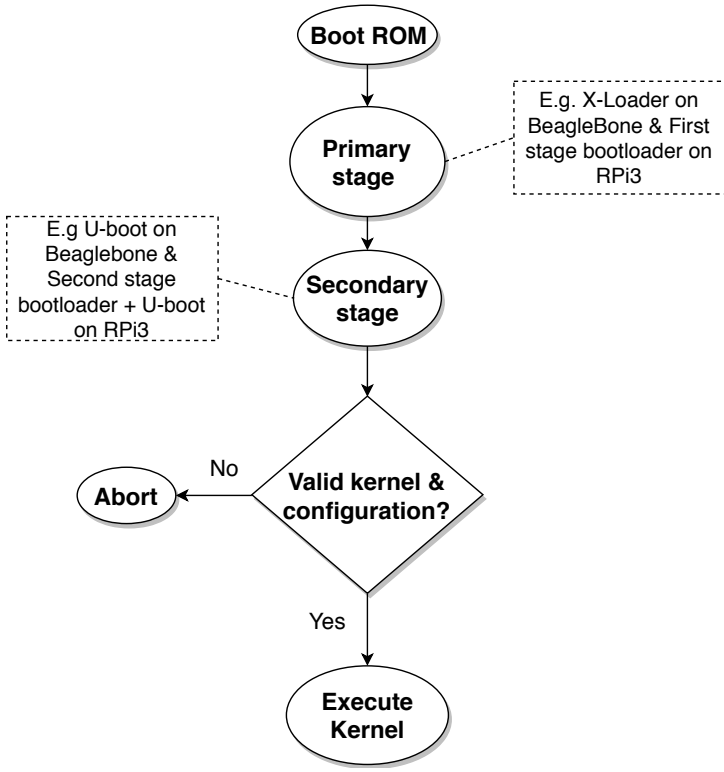


Figure 2.1: The boot process of the medium-scale embedded systems consists of multiple stages. The Boot ROM and first stage boot-loader are hard-coded by the manufacturer and are hard to modify. The second stage and the kernel code are modifiable and usually stored in flash memory.

which is 72% of the boot-loader execution time, and 36% of the entire boot time (5352ms) of an off-the-shelf Linux distribution.

The remainder of the paper is structured as follows. First, Section 2.2 introduces basic concepts and definitions of secure boot. Next, Section 2.3 presents our threat model. We present secure boot on off-the-shelf embedded hardware in Section 2.4 and evaluate its performance in Section 2.5. In Section 2.6 we discuss the limitation of secure boot in IoT devices. Section 2.2 presents related work and Section 2.7 concludes the paper.

2.2 Background and Definitions

In this section, we introduce the required background for both the boot process of a medium-scale embedded device and secure boot in particular.

Boot Process. Commonly, manufacturers of embedded devices divide the boot-up process [55] into several stages (see Fig. 2.1). The purpose of each stage is to prepare the CPU and transfer code fragments from external to internal memory. Eventually, the boot-loader loads the operating system and starts the execution of the kernel. The boot process begins with the Boot

ROM, provided by the manufacturer, which sets and initializes the peripherals. The Boot ROM prepares the system to execute the primary boot stage. The primary stage configures the system and prepares the memory for loading the secondary stage boot-loader. The secondary stage is the actual bootloader of the operating system.

Commonly, both the Boot ROM and the primary stage are tamper-proof, as both are hard-coded in the device firmware. However, manufacturers allow the modification of the second stage to provide flexibility and support for different bootloaders and operating systems. As a result, a secure boot process has to ensure the integrity of the kernel and application code before executing the secondary stage.

Security Challenge. The code in each stage can change the overall status of the system and often the next-stage software. As a result, we cannot trust a self-verified software, as it can be modified to provide a false verification status. Therefore, we need to verify in advance, and before we run each piece of software that we give control over the system.

Secure Boot. In a secure boot process, an inherently trusted component triggers the boot process, which is a tamper-proof component referred to as the *Roots of Trust* (RoT) [56]. The Trusted Computing Group (TCG) [56] defines RoT as a set of functions designed to be trusted by the operating system. In embedded systems, RoT can be the Boot ROM (see Fig. 2.1), which verifies the next-stage software and executes only authentic software. Each stage verifies the integrity of the next one leading to a *Chain of Trust*. For the verification, we can use a dedicated monitoring hardware co-processor. TCG has defined an international standard called the Trusted Platform Module (TPM), which defines the properties that those modules need to fulfill.

We note that different standards and vendors use various terminology to describe a secured boot process: Common terms include, for example, Secure boot, Trusted Boot and Verified boot. Different solutions have been defined and implemented in specific environments including personal computers, data centers, routers, and mobile phones [28, 51, 57]. Especially, *Secure Boot* has been among the standard techniques to define a secured process to assure the integrity of each booting steps [20]. In table 2.1 we compare the terminology for the existing techniques.

Related Work. Recent works demonstrate the need for securing the boot process of connected devices from consumer level printers to industrial robots [31,32]. There is a large body of research in the field of securing and verifying the boot process. Khalid et al. [26] discuss the difference between secure and trusted boot and further evaluate the performance overhead using FPGA boards. Liu et al. give a slightly different approach for system verification. [58] and Lebedev et al. [27], where they are giving examples of a remotely attested system using FPGA embedded systems. In contrast to our work, they focus on FPGA embedded systems while this paper focus on embedded IoT systems. From the practitioner's side, Google's Chromium OS uses verified boot, which builds a chain of trust [28]. For recent work regarding IoT devices, Asokan et al. [29] focus on solutions regarding the firmware update on large-scale IoT deployments. For constraint devices, Boot-IoT [30] propose an authentication scheme towards secure bootstrapping.

Table 2.1: Terminology comparison

Term	Halt	RoT	Verification	Added HW
Secure boot [26]	Auto-termination	Boot ROM	By certificate authorities (Remote attestation)	Not specified
Trusted boot [51]	Letting users decide	Boot ROM	Compare hash values	HSM
Verified boot [28]	Letting users decide	Boot ROM	Stored cryptographic hash comparison	Not specified
Measured boot [59]	No termination	BIOS	Measures hash of objects and logs them	Not specified

2.3 Adversary Model

In this section, we discuss attack vectors on the boot process of IoT systems and introduce our adversary model. A medium-scale embedded device commonly consists of the application itself, an embedded operating system, and the boot firmware. The boot firmware is similar to a BIOS in commodity computers and manages the initial boot process, as discussed in Section 2.2. From a security perspective, a secure boot process has to ensure the integrity of each of these components, i.e., that none of them has been modified maliciously [60]. In the context of connected embedded devices, i.e., IoT devices with Internet connectivity, for example, via 5G/LTE, Bluetooth or WiFi, this leads to two main directions of attack: (1) the traditional attack vector of gaining physical access to the device and (2) adversaries can manipulate the firmware via their Internet connection. Thus, connectivity opens new attack vectors, when compared to traditional embedded devices without connectivity.

To compromise a connected IoT device, an adversary may use security holes in both operating systems and its applications to trigger execution of remote, malicious code. Via this code, an adversary can potentially modify data [61], the OS [62] and also the boot process [31]. The adversary's goal is to make a permanent malicious modification, which is unobservable to security analysis. Moreover, we argue that for most IoT devices connectivity is essential for their operation, i.e., they cannot provide their services to the users without connectivity. Thus, just disabling Internet connectivity to close this attack vector is not an option for the vast majority of applications. Via physical access, the adversary can directly manipulate and modify the application, OS, and the boot process. The TPM is also exposed by an adversary with physical access, ongoing research by using Physical Unclonable Functions (PUF) [63] is a promising solution. In our threat model and further system design, we focus on the new attack vector that Internet connectivity brings.

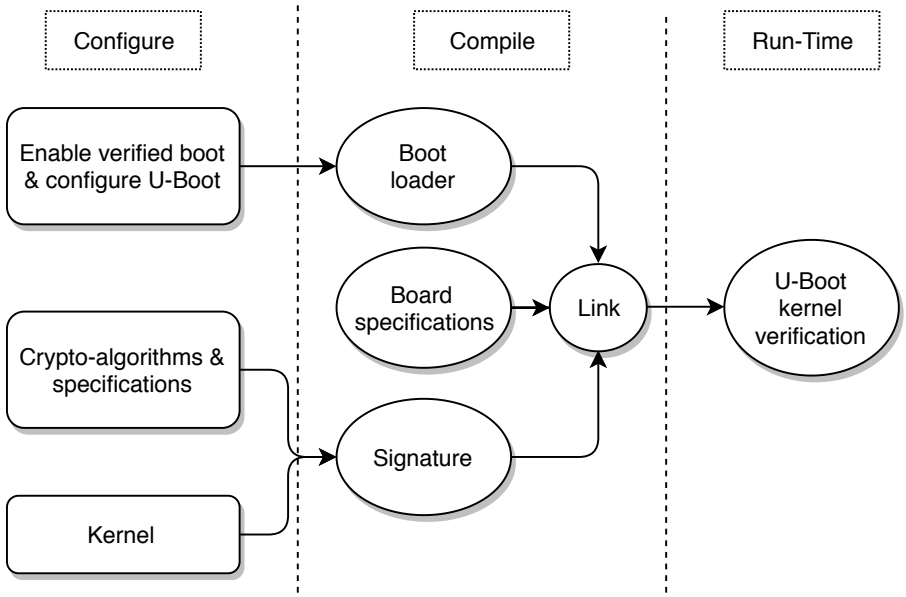


Figure 2.2: From U-Boot configuration to deployment: First, we configure the U-Boot to include the verification module. Next, we link the object files to produce the secure version of U-Boot. Finally, we deploy the executable file on the platform.

2.4 Systems Overview

In this section, we present and discuss two system designs to secure the boot process of an IoT device equipped with an operating system such as embedded Linux: one design is based solely on software mechanisms, and one additionally utilizes hardware primitives. Both designs have specific trade-offs regarding complexity, overhead, and system cost. Both approaches are established [26, 28, 51] in the field, and we do not claim their novelty. Instead, the contribution of this paper lies (1) in comparatively evaluating the overhead that both add to the boot process and (2) in contrasting this overhead to the security each design provides.

2.4.1 Software-based Secure Boot with U-Boot

For the software-based method, we rely on the U-Boot [64] bootloader to verify the integrity of the operating system. In our system design, we make the following assumptions. Firstly, the pre-boot environment of U-Boot has to be trusted, meaning that the security of the boot stages before U-Boot cannot be modified. Typically, the manufacturer embeds the first stage in the Boot ROM. Secondly, U-Boot has to be placed in read-only memory since there is no prior verification of the booting process. Lastly, this design requires read-only storage of the cryptographic hashes used to verify the integrity of the operating system.

U-Boot divides the security process into three steps: *Configure*, *Compile*

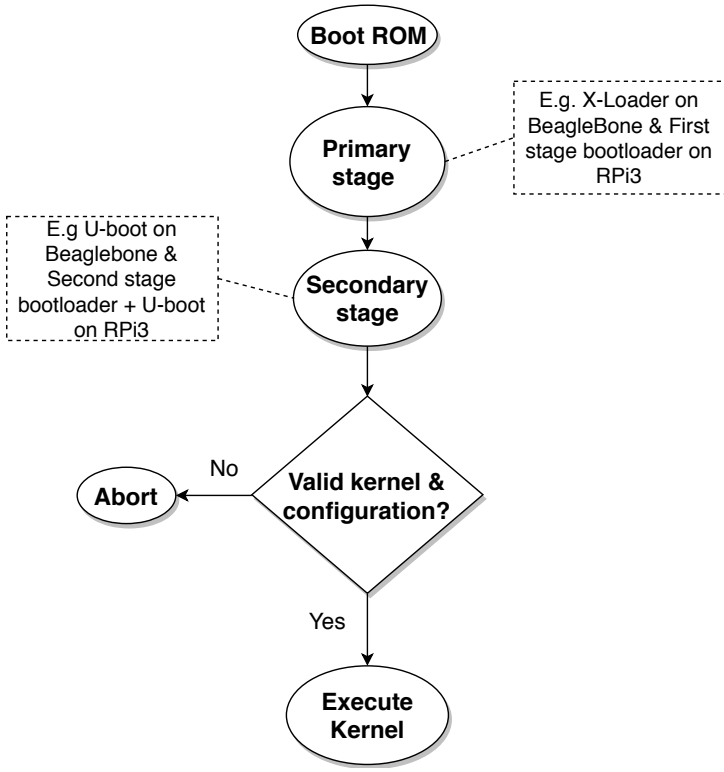


Figure 2.3: Secure boot sequence with U-Boot: U-Boot runs as the second stage, which verifies the kernel and the chosen configuration. U-Boot passes the control of the system to the operating system only after successful verification.

and *Run-Time*, see Figure 2.2. The software-based secure boot process extends the boot process with an additional verification step, see Figure 2.3. As a result, the bootloader only boots the operating system once it has successfully verified the integrity of the operating system. In practice, U-Boot binds the kernel with the hardware information of the board. Thus, U-Boot verifies that the kernel is correct and it will run on the specific hardware configuration.

To verify the integrity of the kernel efficiently, we need to resolve the digital block data of each image to a single value; a conventional method is to use cryptographic hash functions [19]. Cryptographic hash functions map an arbitrarily long data to a small and fixed output, but they need to fulfill specific properties to be considered cryptographically secure [19]. U-Boot supports three cryptographic hash functions, namely MD5, SHA-1 & SHA-256 [19]. The hash functions have the following digest sizes: (1) MD5: 128-bit (2) SHA-1: 160-bit (3) SHA-256: 256-bit. Finally, the hash digest is being signed by the private key, and the bootloader (U-Boot) can verify the authenticity of the hash value by applying its public key. Various public key algorithms could verify the hash digest of the image. Popular public key cryptographic algorithms are RSA [19] and Elliptic Curve Cryptography (ECC) [19]. U-Boot currently supports only RSA, and the two supported key sizes are 2048-bit and 4096-bit. The key size is the critical factor of public key algorithms; bigger key sizes are

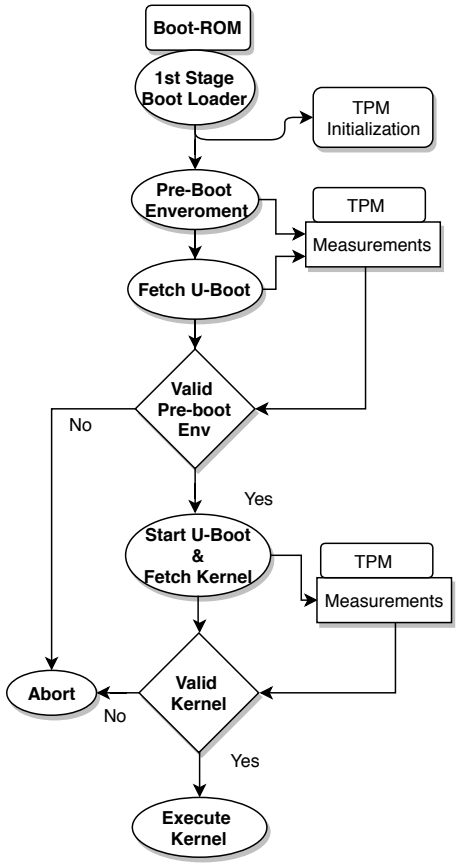


Figure 2.4: Secure boot with a TPM co-processor: The manufacturer provides the boot-ROM and first stage bootloader. We split the second stage into two phases: 1) The pre-boot environment, where we check the integrity of U-Boot. 2) U-Boot execution, where we check the integrity of the operating system.

more difficult to break. On the other hand, the larger the key size, the longer the verification takes [19].

2.4.2 Hardware Security for Secure Boot

After introducing the software-only secure boot process, we next introduce secure boot with a hardware security module. This design further increases the security of the boot process at the cost of adding additional hardware and boot latency.

This design is based on the TPM module as proposed by Khalid et al. [26]. The method always starts with the initialization of the TPM, which ensures that the TPM is activated. The TPM provides the following functions defined by the standard [65]: (1) **Measurement**, TPM calculates the hash of the input data using SHA-1. (2) **Extend**, TPM takes the current hash-value inside the register, appends the Measurement and produces a new hash value (3) **Control Transfer**, the TPM passes the system control to the successfully

Table 2.2: Hardware specifications

Model	**Hardware**
Raspberry Pi 3 Model B	ARM® Cortex A53 - 1.2GHz(quad-core) 1 GB LPDDR2 RAM
Beglebone Black C	ARM® Cortex A8 - 1GHz 512MB DDR3 RAM
Cryptocape (by Cryptotronix)	TPM-Module:AT97SC3204T

verified entity. The process continues by calculating the cryptographic hash value of the boot environment, which includes the system configuration before loading the secondary stage boot loader (see Section 2.2). The process consists of a repetitive **Measure–Extend–Execute** procedure [26]. This method is a common way to ensure a **Chain of Trust** [66], which verifies the integrity of the different stages step by step. The TPM transfers the control of the system to each measured image only if it has successfully verified the extended hash-value. In the case of failure, the boot process will halt as shown in Figure 2.4.

For this technique, we make the following assumption: The first entity of the *Chain of Trust* needs to be trusted, which in this case is the boot ROM and first stage boot-loader. The manufacturer should embed this code in a way so that nobody can modify it, for example, by placing it in read-only memory.

2.5 Evaluation

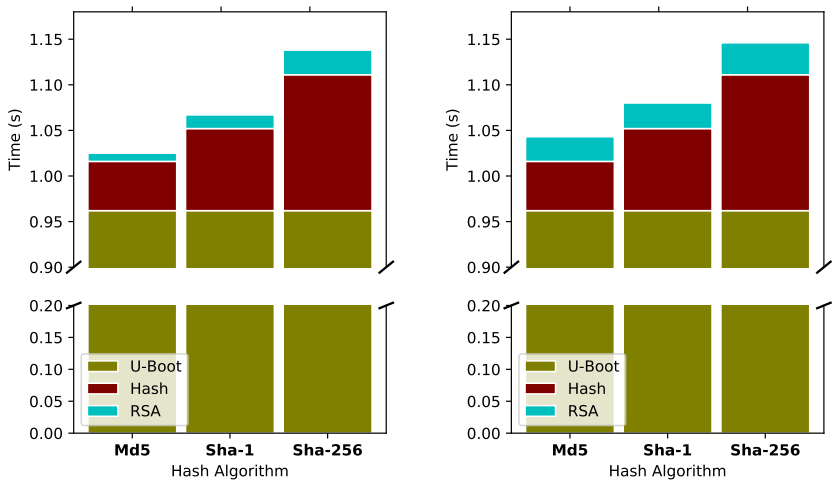
This section evaluates both system designs in detail and focuses on the overhead of the secure boot process in each system. Practical application scenarios motivate this evaluation, for example, vehicles are expected to be immediately usable after ignition, imposing a low-latency requirement between turning on the ignition and the full boot up of all the micro-controllers of a vehicle. Also, home appliances like smart vacuum cleaners and smart heaters experience frequent turn off and on by their users, where the boot time can affect the usability of the device.

2.5.1 Experimental Setup

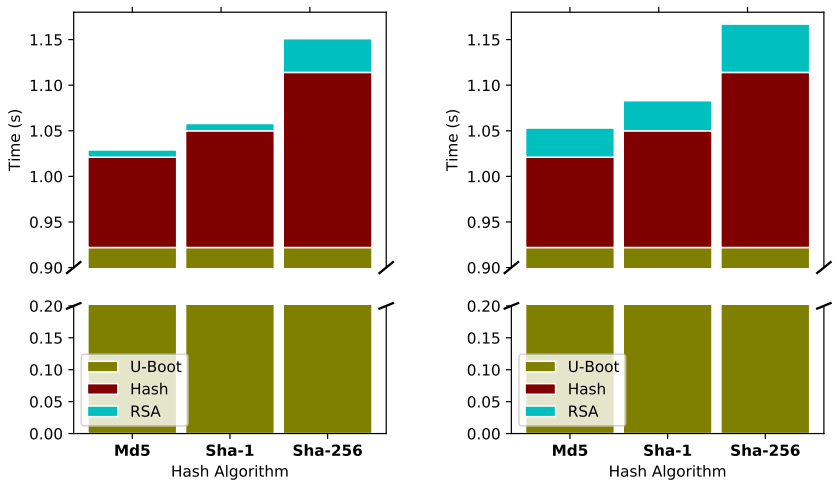
We implement our system design on two generic embedded platforms for IoT applications [67] that are readily available, namely a Raspberry Pi and a BeagleBone (see Table 2.2). Finally, we extend the capabilities of the BeagleBone to support hardware cryptographic primitives.

2.5.2 Evaluation Results

Next, we present the results of our evaluation of the overhead of the secure boot process. We begin with the software-based method and continue with the hardware-based technique. The results summarize the performance of our system design.



(a) RSA 2048-bits key-size on BeagleBone (b) RSA 4096-bits key-size on BeagleBone

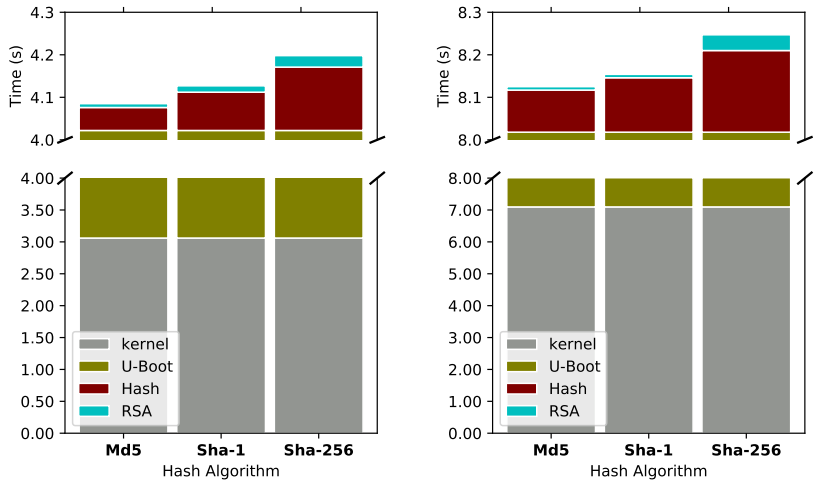


(c) RSA 2048-bits key-size on RaspberryPi (d) RSA 4096-bits key-size on RaspberryPi

Figure 2.5: For the evaluation of U-Boot using the BeagleBone & Raspberry Pi, we apply RSA with a key size of 2048 and 4096 bits. The U-Boot time refers to the performance without the verification module. All figures compare the three available hash functions: MD5, SHA-1, SHA-256. Note the graphs have discontinuing scale numbers

2.5.2.1 Software Mechanism of U-Boot

Figures 2.5a, 2.5b, 2.5c & 2.5d present the overhead of verification using different key sizes and three different hash functions (MD5, SHA-1, SHA-256). The average execution time for U-Boot without any security mechanism is 976ms for the BeagleBone and 903ms for Raspberry-Pi. These numbers form the baseline to compare the overhead of secure boot. The entire boot time of the off-the-shelf Linux-kernel (Debian GNU 7) on BeagleBone is 4s, and for Raspberry Pi (Debian Jessie 4.4) is 7s (see Figure 2.6).



(a) Entire Boot Time: BeagleBone

(b) Entire Boot Time: RaspberryPi

Figure 2.6: Evaluation of entire boot time using the BeagleBone & Raspberry Pi. The RSA key-size is 2048-bits, and we compare with the three available hash functions: MD5, SHA-1, SHA-256. The RSA overhead is very small and barely visible. Note: the graphs have discontinuing scale numbers

We begin evaluating the overhead that different hash functions in U-Boot bring. In this evaluation, we set the key-size of RSA to 2048-bit and use BeagleBone. With the MD5 hash function, the average overhead is 58ms, representing 5.7% of the U-Boot execution time (see Figure 2.5a), and 1.4% of the entire boot time of the Linux kernel (see Figure 2.6a). With the SHA-1 hash function, the average overhead increases to 117ms, which is 11% and 2.8% of the U-Boot execution (see Figure 2.5a) and the entire boot time of the Linux kernel (see Figure 2.6a), respectively. For SHA-256 hash function, the overhead increases to 164 ms, 15% and 4%, respectively (see Figure 2.5a & 2.6a).

Next, we increase the key-size of RSA to 4096-bits, using the same hash functions and BeagleBone. This key-size increases the overhead by 24ms to 35ms depending on the hash function (see Figure 2.5b). For the same experiment using Raspberry Pi, we observe similar results (see Figures 2.5c & 2.5d).

2.5.2.2 TPM Hardware on BeagleBone

Table 2.3 presents the overhead after introducing the hardware primitive (TPM). The initialization of the TPM takes 993ms. TPM uses SHA-1 as the hash function and the overhead of calculating the measurements (see Section 2.2) is 138ms. For extending the Registers (PCR) TPM takes 791ms. Overall, the whole method takes 1923ms, which adds an overhead of 36%.

Table 2.3: Verification overhead of TPM

Overhead	Average time (ms)
TPM Initialization	993
Measurements	138
Extend PCR values	791
Total	1923

2.5.3 Discussion

Configuration choices, like the hash function, have different performance impact and trade-offs. For example, MD5 provides better performance, but it is no longer recommended [68]. FIPS 180-4 Secure Hash Standard (SHS) recommends SHA-1 and SHA-256, but Stevens et al. [69] have found the first collisions on SHA-1.

Regarding the performance of Secure boot with TPM, we have noticed a higher verification overhead (approximately eight times more) compared to the software-based technique (Verified U-Boot). The main reason for this is that there are more measurement requirements in this method: we verify the kernel, U-Boot and boot states which include the complete system configurations. Another source of overhead is the initialization time of the TPM. It is worth to notice that the extra hardware does not intend to accelerate the cryptographic functions, but rather to provide stronger security properties as we explained in previous sections.

To conclude, if we compare the different parts of the boot-time we notice that loading the kernel is the most time-consuming part. Thus, we argue that while the overhead of Secure Boot is not negligible, its overall performance overhead is limited. This performance makes Secure Boot a practical solution to secure the software-stack of medium-size devices in the Internet of Things. For application where boot-up needs to be reduced further, customized, modular OS kernels with application-specific functionality could be an option to improve the boot performance.

2.6 Limitations and Discussion

In this paper, we evaluate the time performance of a software- and hardware-based secure boot techniques. The boot performance, i.e., the time until a device has booted, is a critical aspect, for example in industrial IoT systems like autonomous vehicles. While secure boot ensures system integrity, it cannot protect against all attacks. In the following we discuss key limitations:

Availability. A general limitation of secure boot is the lack of protection against persistent DOS-attack caused by the mechanism. An attacker can try to modify the integrity of the operating systems repeatedly and reboot the system. This attack will prevent the system from booting-up, and it is possible a DOS-attack caused by the security mechanism. We need secure boot techniques that can adapt and recognize such an attack vector. Moreover, this attack highlights the complexity of the security techniques in IoT which involves

heterogeneous devices. For example, a user can still expect a compromised smart-light to work in safe mode. However, a compromised industrial robot which involves safety-critical aspects, it should immediately halt.

Applicability. In this paper, we focus on medium size embedded boards (e.g., ARMv7), where resource constraints do not prevent the extension of the boot process. In small constraint IoT devices (e.g., ARM Cortex M4) similar approaches may not be applicable for several reasons. First, those devices do not separate the boot process in stages. Moreover, we need to implement a bootloader efficient to meet memory and run-time constraints for those devices. Second, the firmware and the application is stored together in flash memory, without any protective barrier. Finally, the limited CPU power makes it hard to make the cryptographic calculation. We can expect a different overhead compare with that of the evaluation in section 2.5. Hardware support like TPM is not available for small IoT device.

Scalability. One of the benefits of the Internet of Things is the ability to upgrade the firmware over-the-air (OTA), i.e., via the Internet connection of the device. This flexibility provides scalability for vendors to upload the firmware and application updates to already deployed IoT devices. However, after each update, the credentials matching the firmware need to be updated. This update is challenging, as many TPM modules do not allow direct updates of credentials to protect against attacks.

2.7 Conclusion

The security aspects of embedded systems become more critical with the rise of the Internet of Things. Secure boot is one of the primary tools to secure IoT applications and their operating system. This paper presents and evaluates trade-offs regarding the implementation and the performance of secure boot. Our results show that the software-based method increases the overall boot-up time by 4%. The hardware-based one adds an overhead of 36%.

For future works, we plan to evaluate the impact of different system configurations and kernel configurations on the performance of secure boot. Moreover, we will focus on the design, implementation, and evaluation of secure boot on smaller and more constrained devices (e.g., ARM M4).

2.8 Acknowledgments

This work was supported by the Swedish Research Council (VR) through the project “AgreeOnIT”, the Swedish Civil Contingencies Agency (MSB) through the projects “RICS” and “RIOT”, and the Vinnova-funded project “KIDSAM”.

Paper B

Chistos Profentzas, Magnus Almgren, Olaf Landsiedel

IoTLogBlock: Recording Off-line Transactions of Low-Power IoT Devices Using a Blockchain

Proceedings of the 44th IEEE Conference on Local Computer Networks (LCN) 2019.

Chapter 3

IoTLogBlock

For any distributed system, and especially for the Internet of Things, recording interactions between devices is essential. At first glance, blockchains seem to be suitable for storing these interactions, as they allow multiple parties to share a distributed ledger. However, at a closer look, blockchains require heavy computations, large memory capacity, and always-on communication to the cloud; these are three properties that are challenging for IoT devices with limited resources.

In this paper, we present IoTLogBlock to address these challenges. IoTLogBlock connects resource-constrained IoT devices to the blockchain, and it consists of three building blocks jointly enabling recording transactions: a lightweight contract signing protocol, a blockchain network, and a smart contract. The contract signing protocol allows devices to interact locally to perform transactions, even if no communication to the cloud and the blockchain exists at that moment. At a later time, devices forward the stored transactions to the blockchain, where a smart contract ultimately verifies the transactions.

We evaluate our design on low-power devices and quantify the performance in terms of memory, computation, and energy consumption. Our results show that a constrained device can create and sign a transaction within 3 s on average. Finally, we expose the devices to network scenarios with edge connections ranging from 10 s to over 2 h.

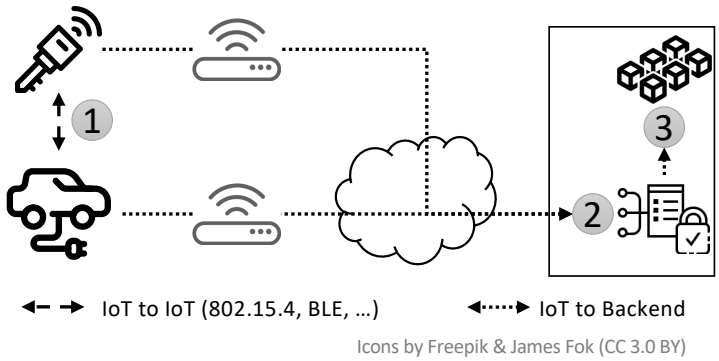


Figure 3.1: Motivating scenario: car rentals where (1) a smart-key and a smart-car perform off-line transactions using a contract signing protocol over low-power wireless technologies. Later, the IoT devices independently forward their transactions via an edge device to the blockchain where (2) a smart contract validates and then (3) stores the transactions immutably.

3.1 Introduction

Blockchains store immutable records of transactions in a so-called distributed ledger. Transactions are generated by parties which do not necessarily trust each other. As records in a blockchain are immutable and can be verified by all parties involved, a blockchain creates the required trust between the involved parties whether a particular transaction is part of the blockchain or not. This makes the blockchain an ideal candidate for storing records of transactions produced by the plethora of connected devices in the Internet of Things (IoT). These devices interact frequently and produce records of interactions which their applications want to store, while two IoT devices commonly do not trust each other, as they are, for example, owned or operated by different entities.

We argue that the following key challenges are overlooked in the context of IoT and blockchains. To perform transactions, blockchain clients need to be (a) connected to the blockchain, (b) often store large parts of the blockchain, and (c) perform heavy cryptographic operations. These are three requirements that today's resource-constrained IoT devices can hardly fulfill.

Recent works demonstrate the use of blockchains for data provenance [38] [39], but there are two significant issues that existing architectures cannot address. First, the integration of resource-constrained devices is limited due to the communication requirements and fixed connection points. For example, in many designs [41] [42] devices need to seek active communication with the network for each transaction, which demands considerable energy consumption. Moreover, in the case of mobile IoT devices such a design demands for complete network coverage through, for example, cellular technologies such as 4G, which in turn further increases cost and energy consumption. Similarly, LPWAN technologies such as LoRa and SigFox do not provide the required bandwidth required by the cryptographic operations of a blockchain. Second, the proposed architectures cannot ensure non-repudiation of the transactions given that stakeholders often have conflicting interests. We argue that it is essential that when two IoT

devices create an off-line transaction, the stakeholders cannot later deny their participation therein. Data tampering and device impersonation attacks [43] raise concerns regarding the secure collection of sensor data.

With this in mind, we propose IoTLogBlock with the following properties. IoTLogBlock combines an (1) **optimistic contract signing protocol** with a (2) **smart contract**, (3) deployed on a **blockchain**, as shown in the motivating car rental scenario in Figure 3.1. The contract signing protocol allows two or more nodes to sign an off-line transaction mutually and achieve non-repudiation [70]. We assume only intermittent network access (due to power conservation or infrastructure issues) and the IoT devices can store the off-line transactions in local memory while waiting for an edge connection. An edge device forwards the transactions to the blockchain network for validation using a smart contract. The smart contract is similar to a stored procedure of regular databases, and it is triggered upon events sent by the network.

Overall, the combination of a contract signing protocol with the smart contract achieves non-repudiation and enables the integration of resource-constrained IoT devices to a cloud-based blockchain. As a result, IoTLogBlock allows IoT devices, which are connected to the Internet infrequently, to employ blockchains. We make the following contributions.

- We design and implement IoTLogBlock, an open-source architecture[1] for recording IoT transactions using blockchain. IoTLogBlock achieves non-repudiation, avoids dependence on fixed network infrastructures, and ensures a low-power radio duty-cycle.

- We propose a practical solution to implement a contract signing protocol on IoT devices.

- We design and implement a smart contract to act as the online validator for the off-line transactions.

- We quantify the performance of IoTLogBlock in terms of computation, delay, memory, and energy consumption. Notably, we find that low-power devices (TI-CC2538) can create a transaction in 3 s. We further calculate the latency of IoTLogBlock in different scenarios, with an edge connection from 10 s to over 2 h.

- We finally provide a discussion of implementation challenges and trade-offs regarding the integration of resource-constrained devices with a cloud-based blockchain.

Organization. The paper is organized as follows. In Section 3.2, we provide the necessary background regarding our approach. In Section 3.3, we describe a motivating example and our adversary model. In Section 3.4, we provide the system design and highlight security aspects. We evaluate our results in Section 3.5, which is followed by a discussion and a description of related work before concluding the paper.

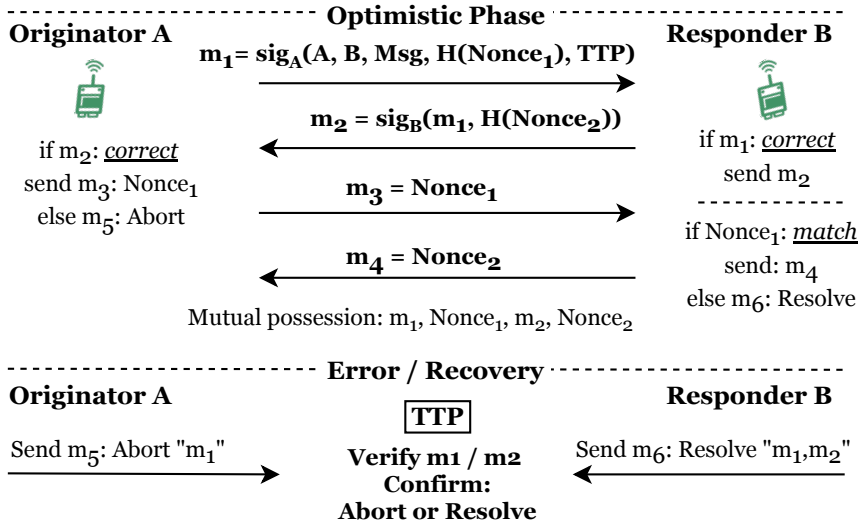


Figure 3.2: Signature exchange in the Asokan-Shoup-Waidner protocol. In the optimistic phase, two nodes agree to sign a contract. At the end of the protocol, both participants have a copy of the mutually signed agreement. In the error/recovery phase, a node tries to abort or resolve a previous round of the protocol using a Trusted Third Party (TTP).

3.2 Overview and Background

We provide the background on the building blocks of IoTLogBlock: contract signing protocols, smart contracts, and blockchain.

3.2.1 Contract Signing Protocols

Contract signing protocols [71] allow two or more parties that do not trust each other to sign a pre-agreed text. These protocols provide fairness, timeliness, and sometimes also an abuse-free property [72]. Two parties (Alice and Bob) exchange their signatures in a fair-way [23], where Alice can obtain Bob's signature only if Bob can obtain Alice's signature and vice-versa. Timeliness [72] means that a participant is not able to make the other wait for an indefinite amount of time. Finally, with the abuse-free property [72], a participant cannot determine the outcome of the protocol.

We can group the contract signing protocols in three categories: protocols with an online Trusted Third Party (TTP) [21], protocols without TTP (e.g., time commitments [22]), and protocols with an off-line TTP (e.g., optimistic [23]). As the first two are computationally demanding, they are not suited for resource-constrained IoT devices. Hence, we focus on the third group of optimistic contract signing protocols. Specifically, we build on the **Asokan-Shoup-Waidner (ASW)** [23] protocol, which provides fairness and timeliness, but it is not abuse-free [72]. The ASW protocol allows two parties

[1] https://github.com/iot-chalmers/IoTLogBlock.git

to exchange signatures as shown in Figure 3.2, without actively invoking the trusted third party. However, an online TTP is available to resolve issues if one of the parties does not follow the protocol (crashes or behaves maliciously). There are three sub-protocols involved in the process. The first is to complete an optimistic exchange of signatures (without the involvement of the TTP). The second is to allow a participant to abort a signature exchange (abort sub-protocol), and the third is to ask the TTP to resolve an incomplete signature exchange (resolve sub-protocol).

3.2.2 Smart Contracts

A smart contract is a digital representation of an agreement between two or more parties, written as an event-based program and stored immutably on the blockchain. The blockchain assigns a public key to each smart contract, and applications can use these keys to send a transaction to the blockchain, which triggers the execution of a particular smart contract. Each smart contract has its internal state to keep events, and the execution may add a new record to the blockchain. Finally, as described in [40], a smart contract can act as an escrow to resolve (dis)agreements between two or even multiple parties. In IoTLogBlock, we utilize this capability and employ the smart contract as off-line TTP to resolve issues of the contract signing protocol. As the code and transactions of the smart contract can be verified by everyone involved, it inherently gains trust as a third party.

3.2.3 Blockchains and Hyperledger

A blockchain is essentially a distributed ledger: multiple nodes replicate blocks of records as an append-only data structure. Each block has multiple records which are linked together into a chain using the cryptographic hash value of the previous block. Any change to an older record of a block inherently invalidates all recent blocks and their records. The ledger is replicated over multiple nodes in the network to provide fault tolerance, and the replicas reach a consensus regarding the order in which they add blocks to the chain.

Conceptually, we can categorize blockchains as public or private. The former is the initial design of Bitcoin [2], where any node can join the network and make commits. The latter enforces access control lists to determine who can participate in the consensus process and make commits.

An example of a private blockchain is Hyperledger Fabric [42], which is a collaboration between Linux Foundation and IBM to provide an open-source distributed ledger. The platform has three main components (see Figure 3.3). First, a *Membership Service Provider (MSP)* manages the identities on the permission-based blockchain [42]. The MSP authenticates and authorizes peers that can make changes to the blockchain. Second, multiple authenticated *peers* participate in the network to maintain the ledger and run the smart contracts. Finally, the *Fabric client(s)* is an authenticated node that allows the blockchain network to interact with the outside world.

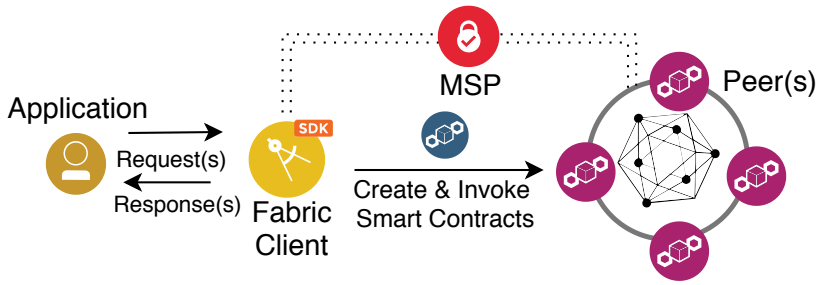


Figure 3.3: The Hyperledger Fabric consists of: 1) the Membership Service Provider (MSP), which issues the access list of participants, 2) the private network of peer(s) that maintain the blockchain, and 3) the Fabric client which provides the application interface.

3.3 Application Scenario and Adversary Model

This section discusses a motivating example and outlines our adversary model for IoTLogBlock.

3.3.1 Application Scenario

As a motivating scenario, we consider a car rental service with regular customers [73]. The rental company provides cars at a service point or distributed throughout the city's parking lots. The customer signs up and receives a smart token (key) that will open a car at any time. The company wants to ensure that they know what customer used a car during which period, to charge the customer for the use and also be able to charge for potential abuse. Customers, on the other hand, want a well-functioning car, and they are only charged for their specific use based on their agreement with the company.

In the general case, the stakeholders are known, but they have conflicting interests. For example, the driver would like to pay as little as possible while the car rental service would like the highest fee possible. It is also likely that situations (accidents) will occur where it is essential to keep an immutable ledger of the transactions between the stakeholders. The ledger can then be used afterwards to analyze misbehavior or malicious activities. Finally, we argue that even though a modern car can afford LTE or 4G connections for recording its activities, it will still benefit from our design as cellular coverage is not always available, especially in rural areas.

3.3.2 Adversary Model

We assume three potential threats as part of the adversary model. First, an adversary may try to exploit insufficient data auditing of a transaction (e.g., an IoT node skips to report a transaction), where the log does not capture enough data to determine the event of a transaction. This is a repudiation threat where, for example, a car or a key may create an off-line transaction in such a way that there is no proof that an online validator can verify the

event of the off-line transaction. Second, we consider an adversary that may try to exploit weak audit mechanisms, including attempts to destroy the audit mechanism (e.g., blockchain) of the transactions. This is a scenario where a customer has used a car and tries to remove the events to claim the money back or not pay in the first place. Third, an adversary may try to include data from an unknown source or untrusted device. As a result, the system log may include data from unknown devices. Here the register entry leads to an unknown customer, and the log system is unable to connect a transaction with a real customer.

3.4 System Design

3.4.1 Design Overview

There are three key building blocks to IoTLogBlock (Figure 3.1): (1) the interaction between two IoT devices using a contract signing protocol, (2) the verification of the resulting transaction between the devices using a smart contract, and (3) the final immutable storage of the result on the blockchain for all participants (auditing, finding misbehaving nodes).

To prototype IoTLogBlock, we also used several existing technologies to tie it together. Below we describe the full system and mark where our contribution lies with a (*) and the corresponding number in Figure 3.1. These steps will then be further elaborated below.

3.4.1.1 Node discovery and node interaction (1*)

When two low-power wireless devices shall interact, for example, when the smart key attempts to open the car via wireless communication, the two devices first have to discover each other. In our prototype, the nodes are synchronized with an established local and autonomous communication protocol, TSCH [74], and they do not depend on any central entity. Next, they have to perform a wireless transaction, where at the end the car grants access (or not) to the smart key. Here our contribution lies in the combination of established low-power wireless protocols, namely TSCH and 802.15.4 [75], with an optimistic two-party contract signing protocol, which is further described in Section 3.4.3.

3.4.1.2 Interaction between nodes and the cloud

Sporadically, an IoT node will come into range of an edge device with cloud connectivity. In this case, the IoT node utilizes the edge device as a relay to transfer the off-line transactions into the cloud for verification and storage. We assume that the edge device has robust capabilities, and it can establish secure connections (e.g., SSL) with a node of the blockchain network (see Figure 3.1). This step builds on established mechanisms and is not further described in the design of IoTLogBlock.

3.4.1.3 Verification and storage in the cloud (2*, 3*)

Through the edge device, all transactions reach the cloud, where an authenticated node/peer runs a smart contract to verify each transaction, detecting

misbehaving nodes, and finally to store each transaction. The smart contract validation is described in Section 3.4.4 and the final step, permanent storage, is described in Section 3.4.5. We conclude in Section 3.4.6 with the security analysis of potentially misbehaving nodes.

3.4.2 Setup: Deploying new Devices

When we add a new device to the system, i.e., we deploy a new smart key, this device creates a private/public key pair. Each device will use its private key to sign its transaction. The public key of each device is stored in the blockchain and is used to authenticate clients later and verify their transactions. Moreover, we also employ a smart contract to manage the list of registered devices.

3.4.3 Creating and Signing Transactions

Each device maintains a sequence number that uniquely identifies each of its transactions, by simply incrementing a counter for each new transaction. The sequence number is later used for verification by the smart contract. In IoTLogBlock, a transaction is created as follows: Once two devices have discovered each other, they exchange essential information such as their IDs and the transaction upon which they want to agree. Next, they sign and exchange a message containing their IDs, current local sequence numbers, and the transaction itself, following the optimistic two-party contract signing protocol (see Section 4.2). If the transaction completes, each device has a transaction signed with the private key of both parties that states the IDs of the participants, i.e., their public keys, their respective sequence numbers, and the transaction content itself. Both IoT nodes individually upload this information into the smart contract via an edge device once they come within range. As transactions are signed, manipulations of these, for example, at the edge device are not possible.

3.4.4 Validation with the Smart Contract

IoTLogBlock deploys a smart contract in the blockchain to act as a validator for the received transactions from IoT devices. Algorithm 1 shows an abstract pseudo-code version of the validator. The algorithm starts by checking the validity of the devices' signatures and reports any misbehavior. As a result, transactions will only be committed when signed by registered devices. Finally, the smart contract resolves any conflict by using the sub-protocols of the ASW contract signing protocol [23], as we discuss next.

Abort sub-protocol: If a node crashes, disconnects or misbehaves during the execution of the first half of the contract signing protocol, the abort sub-protocol is invoked. Thus, if there is a timeout or the signature verification fails, the participant sends an abort request to the smart contract by signing a new message which includes the original message_1 of the protocol (see Figure 3.2). If the smart contract has not received a request to resolve an uncommitted transaction (see the resolve sub-protocol below), it confirms the request and registers the abort message to the blockchain.

Resolve sub-protocol: If the two nodes have already exchanged the first two messages of the protocol, before one of the nodes crashes or disconnects,

Algorithm 1: Smart Contract - Validator

```
   /* Error checking, such as checking for key revocation, is
      omitted for brevity.                                      */
   Data: Transaction
 1 validTransaction = True;
 2 forall NodeID in Transaction do
 3 |   Pubkey = RegisteredDevices(NodeID);
 4 |   Status = ECDSA.Verify(NodeID, PubKey, Transaction);
 5 |   if Status is not valid then
 6 |   |   report NodeID;
 7 |   |   validTransaction = False ;
 8 |   end
 9 end
10 if validTransaction is True then
11 |   if Pending_resolve_request then
12 |   |   run Resolve-SubProtocol(Transaction);
13 |   else if Pending_abort_request then
14 |   |   run Abort-SubProtocol(Transaction) ;
15 |   else
16 |   |   record Transaction;
17 end
```

the resolve sub-protocol is invoked. As the nodes have made some progress, the signature exchange can be resolved. The node sends a resolve request containing the first two messages (m_1,m_2) of the protocol (see Figure 3.2) to the smart contract. The resolve sub-protocol is also invoked when a node tries to abort even if it has already sent $message_2$.

3.4.5 Register Transactions in the Blockchain

After the transactions are validated by the smart contract, they are appended to the blockchain. By its very nature, this leads to a distributed ledger open to all stakeholders. Transactions stored in the blockchain are immutable and cannot be changed without invalidating previous transactions, as explained in Section 4.2.

3.4.6 Security Analysis

In IoTLogBlock, we focus on the following misbehavior:

1) Bypass sequence numbers: A node could skip or reuse sequence numbers to hide its misbehavior. However, when a node uploads its transactions, the smart contract detects any duplication or gaps in the sequence numbers. Since the ledger in IoTLogBlock is available to all stakeholders, it is trivial to detect this behavior.

2) Hide transactions: A node could execute a transaction with another node and not report it or even try to delete it. By using the contract signing protocol, both parties have a copy of a transaction. If the other node behaves correctly, it will upload all the transactions, and the smart contract detects the misbehaving node. Thus, a transaction will only go unnoticed if both

involved parties are misbehaving or both fail to establish an edge connection. We argue that it is very uncommon for two parties to collude in this way, as they commonly would not share the same connection or the same goals. For example, while the smart key, or more precisely, the corresponding users might be interested in driving a car for free, the smart car has precisely the opposite interest. Finally, the blockchain ensures the immutability of its records, and a node cannot delete already stored transactions.

3) Unregistered device: An untrusted device can try to overcome the contract signing protocol, and send unsigned or invalid transactions to the blockchain. However, this is taken care of by the protocol itself and the trusted party (here the smart contract), which accepts transactions only by registered devices. In case of a transaction violation, a node can then invoke the sub-protocols to abort or resolve a transaction. Since a node reports all the abort attempts to the blockchain, it is trivial to detect a node that abuses the protocol and block it in the future.

3.5 Experimental Evaluation

IoTLogBlock includes low-power IoT devices, edge computing, and a cloud service (Hyperledger). In our evaluation, we focus on the IoT devices; the edge device plays a secondary role in our architecture, and Hyperledger Fabric has previously been evaluated in detail [42].

Goals. With this section, we answer the following questions: a) is IoTLogBlock technically feasible on low-power IoT devices? b) what is the overhead in terms of computation, memory, and energy consumption of the contract signing protocol? c) what is the overall performance of IoTLogBlock?

Outline. We divide the evaluation into three parts. First, we discuss our implementation of IoTLogBlock and our choices in terms of IoT hardware, software stack, edge computing, and blockchain. Second, we evaluate our implementation of IoTLogBlock in terms of run-time performance, energy consumption, and memory consumption. Third, we present our overall system performance including 1) the time it takes for a sensor node to register a record on the blockchain, and 2) the reliability and limitations of IoTLogBlock when creating periodic off-line transactions.

3.5.1 Implementation and Experimental Setup

IoT Software Stack. We implement the contract signing protocol in C as an application for Contiki-NG [76]. Our implementation employs the Elliptic Curve Digital Signature Algorithm (ECDSA) using the recommended NIST-P 256-bit curve. All the ECDSA operations are performed by the cryptographic hardware support of our target platform. For communication between IoT nodes and to edge devices, we use the TSCH protocol [74] readily available in Contiki-NG. TSCH employs radio duty cycling and provides us with a robust and energy efficient communication substrate, tailored to resource-constrained and potentially battery-driven IoT devices. We would like to note, that the design of IoTLogBlock is not bound to a particular low-power wireless protocol and would also support, for example, BLE.

IoT Hardware and Platform. We target sensor nodes equipped with cryptographic hardware support such as the TI-CC2538 SoC. The SoC contains a 32-bit ARM Cortex M3 CPU at 32 MHz, 32 KB of RAM, and 512 KB of ROM. Moreover, the SoC features a cryptographic engine at 250 MHz and a 802.15.4 radio transceiver. This SoC is common in the community for resource-constrained IoT devices and readily supported by numerous operating systems. In particular, we use the OpenMote platform [77], which combines this SoC with further sensors and a battery.

Edge Devices and Blockchain. In IoTLogBlock, the edge devices receive transactions from the IoT device and upload them to the smart contract of the Hyperledger instance via the Fabric API. For these connections, we use Python and Node.js scripts while we implement our smart contract in GO. Our edge devices run a standard Linux. For simplicity, we deploy the edge devices and the Hyperledger Fabric network using Docker containers on a desktop machine, which features an Intel Core i5 at 2.3 GHz and 16 GB of RAM.

Source Code. We provide our implementation of IoTLogBlock as open-source code in a public repository.[2]

3.5.2 Evaluation of IoTLogBlock on the IoT Node

As the first step, we evaluate our contract signing protocol in terms of memory, computation, and energy consumption. For the memory footprint we use the tools **arm-none-eabi-readelf** and **arm-none-eabi-size** on the binary files. For the computation and energy performance, we rely on **Contiki's Energest** module with a 30 µs resolution timer to measure the duration of each task, log the power modes of the system, and derive the energy consumption. For the values of the electric current (see Table 3.3) we rely on the CC2538 data sheet [47] and a previous evaluation [34].

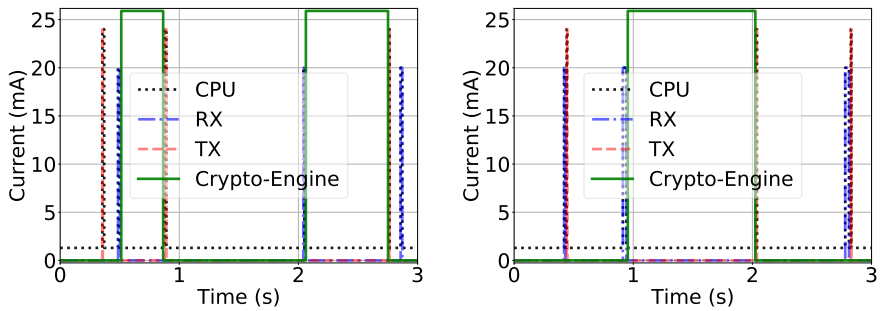
Memory Footprint. Table 3.1 presents the static memory allocation of our implementation, divided into three parts: (a) the operating system (Contiki-NG) with the network stack, (b) the cryptographic library for CC2538, and (c) the implementation of the contract signing protocol (Application). The operating system has a significant impact and consumes 37% of the available RAM. The application combined with the cryptographic library consumes 17% of the available RAM. In total, we consume 54% of the available memory. This leaves 46% of the available RAM to be dynamically used at run-time by the stack and for the temporary storage of off-line transactions, which we evaluate later. Finally, the whole program consumes only 12% of the ROM.

Performance. Next, we evaluate the performance of the cryptographic functions. All cryptographic operations are performed by the cryptographic engine running at 250 MHz, and in Table 3.2 we present the performance of each task. The average time to complete all cryptographic functions of the contract signing protocols in IoTLogBlock is 2.1 s. The most burdensome task – concerning the performance in time – is the signature verification, which takes 715 ms per node. Our protocol uses the SHA256 hash-function multiple times during a single transaction, but the impact is low. The cryptographic operations combined with the latency of the wireless TSCH protocol stack (see Figure 3.4) gives us a total of 3 s per transaction. We argue that this overhead,

[2] https://github.com/iot-chalmers/IoTLogBlock.git

Component	RAM Bytes	RAM Percent	ROM Bytes	ROM Percent
Contiki-NG OS	11,394	37%	40,527	10%
Cryptographic library	1,520	4%	10,775	1%
Application	4,201	13%	7,993	1%
Total footprint	17,115	54%	59,295	12%
Available memory	14,885	46%	452,705	88%

Table 3.1: Memory Footprint of the IoT application (considering max size) on CC2538, which facilitates 32KB of RAM and 512 KB of ROM.



(a) Originator: This node starts the contract signing protocol.

(b) Responder: This node responds accordingly.

Figure 3.4: The electric current drawn by a complete round of the contract signing protocol.

while significant, is feasible for many applications. For the application scenario of the car rentals, it means a user will need three seconds to unlock a car, which we consider practical.

Energy Consumption. Focusing on energy efficiency, we assume that nodes have discovered each other, which is a functionality provided by the TSCH protocol known as TSCH synchronization. This discovery happens quickly, and it has been evaluated previously [74]. In Figure 3.4, we depict the flow of the electric current (in mA) during a complete round of the contract signing protocol, including wireless communication and the use of the cryptographic engine.

In this example, a node (originator) initiates a transaction with a second node (responder). The protocol begins with the originator and responder exchanging a hello message to indicate their availability, visible at time 0.5 s in Figure 3.4. Next, the originator starts the protocol by signing the first message, which takes 0.3 s (see Table 3.2). The responder is waiting in low-power mode for the signature of the originator, which it receives at time 0.9 s. Immediately, the node starts the verification and signs the message (in case of a correct signature). This process takes 1 s, while the originator waits in low-power mode. At time 2.1 s, the originator receives the signature from the responder and starts the verification, which takes 0.7 s. Next, at 2.8 s the originator (in case of a correct signature) replies with its secret nonce value, followed by a reply of the responder.

Function type	Time
ECDSA-Sign on originator	350 ms
ECDSA-Verify on responder	715 ms
ECDSA-Sign on responder	350 ms
ECDSA-Verify on originator	715 ms
SHA256-Hash function	1 ms
Total time	2131 ms

Table 3.2: Performance of cryptographic functions using the CC2538 cryptographic engine running at 250 MHz.

State	Time [ms]	Current [mA]	Energy [mJ]
Cryptographic Engine	1,065	25.9	57.9
TX	32	24	1.6
RX	836	20	35.1
CPU @ 32 MHz	38	13	1.1
CPU @ LPM2	1,029	1.3	2.8
Total	3,000		98.5

Table 3.3: Derived energy consumption of running the contract signing protocol on the CC2538 SoC given a supply voltage of 2.1 V. Note we configure Contiki-NG to use the low-power mode 2 (LPM2) [47].

In Table 3.3, we report the total energy consumption (in mJ) of the protocol, which also includes the wireless protocol stack. We notice that the cryptographic engine contributes significantly (58%) to the total energy consumption. The wireless communication via the TSCH protocol contributes 35%. In total, a transaction consumes 98.5 mJ. We observe that the cryptographic computations and radio communication are the two main drivers of energy consumption in IoTLogBlock.

By default, OpenMote is powered by two standard AA alkaline battery of 2500 mAh. Assuming a self-discharge of 20%, we can utilize 80% of the capacity, i.e., 2000 mAh [78]. As a result, we can expect 10,000 Joules of energy from the cells, which allows IoTLogBlock to perform roughly in the order of 100,000 transactions. While this is merely an estimate, we argue that this order of magnitude of transactions is practical for a wide range of application scenarios, including ours of a battery-powered smart key.

Reflecting on previous work [34] [36], we conclude that, while the cryptographic module consumes the largest share of the energy, the design of IoTLogBlock would not be feasible without it. Implementing the cryptographic functions on the main CPU and without the module would significantly extend the computation time. As a result, the cryptographic handshake would get impractically long and consume even more energy.

3.5.3 System Performance of IoTLogBlock.

We evaluate the overall system in terms of a) local storage capacity b) delay of registering a transaction to the blockchain, and c) the reliability of nodes when creating off-line transactions. In the experiments, we utilize the remaining available RAM for storing off-line transactions while waiting for an edge device to come into range. We collect our results after sending at least 200 packets, and we report the standard deviation when it is not negligible.

We provide edge connectivity to the IoT devices within a period of 1, 10, 100, 1,000, and 10,000 s. These experiments expose the devices to different types of scenarios, ranging from a dense topology with access points to a much more sparse network topology with little edge connectivity. As evaluated previously, a transaction takes 3 s. Thus, in our evaluation, two nodes generate transactions every 3, 30, and 300 s (see Figure 3.5).

Transaction Storage Capacity. We now present how many transactions a device can store until it connects again to an edge device. This is essential, as a device has to drop previous transactions or refuse new ones once its temporary storage is full. Figure 3.5a shows that with edge connections in the same order of magnitude as the transaction generation rate, not much memory is used. As the edge connections get more sparse compared to the transaction generation rate, the devices start to utilize more of the storage capacity, and they reach full utilization of their memory when we provide edge connections in terms of hours (10,000 s). After this point, we need to start dropping records or refuse new ones, which affects the reliability (or availability) of the system (see Figure 3.5c and discussion below).

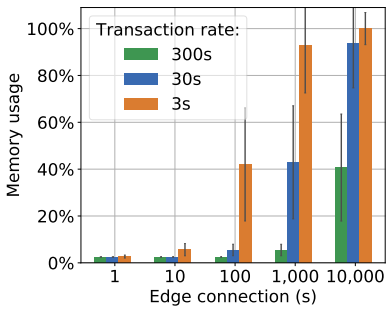
Register Delay. In Figure 3.5b, we present the total number of seconds (delay) that it takes to create, sign, and register a transaction to the blockchain. We can see a correlation between the delay and the periodicity of the edge connection.

System Reliability. We have counted the total amount of attempts a device tried to create a transaction. We quantify the reliability IoTLogBlock as the percentage of the successfully stored transactions on the Hypeledger, which is presented in Figure 3.5c. We notice that the reliability highly depends on two factors: the periodicity of the edge connection, and the transaction generation rate.

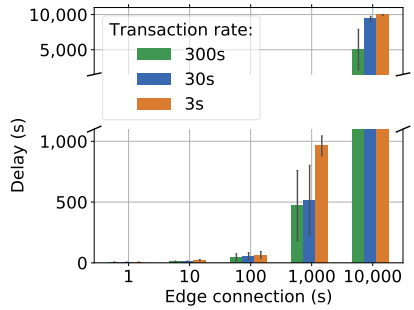
3.6 Discussion and Limitations

In this section, we discuss the benefits, limitations, and remaining challenges regarding our approach.

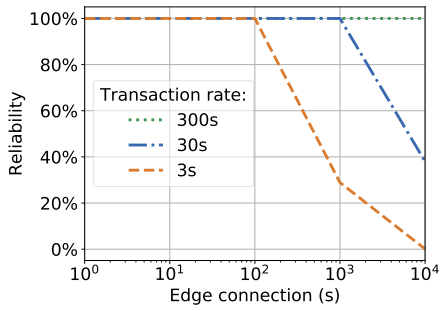
Resource Constraints. The performance of the contract signing protocol (a full-round) averages 3 s. We argue that this demonstrates IoTLogBlock to be functional in practice. Moreover, the protocol utilizes radio and CPU duty cycling for low-power consumption (see Table 4.4). However, the energy consumption of the cryptographic engine demonstrates that security comes with a price. We note that the cryptographic operations sign and verify are essential to any transaction flow of the Blockchain. Moreover, they must be performed on the IoT devices themselves and cannot be delegated to the potentially untrusted edge or cloud services. Thus, we argue that this energy



(a) Memory usage for storing off-line transactions in the local memory of CC2538. The black vertical line shows the standard deviation.



(b) The delay between creating an off-line transaction from IoT devices and registering it to the Hyperledger Fabric. The black vertical line shows the standard deviation.



(c) We count all the transaction attempts, and we quantify the system reliability as the percentage of the successfully created and stored transactions to Hyperledger. When the device memory is full, the devices refuse to create further transactions.

Figure 3.5: We create transactions every 3, 30, and 300 s. The IoT device offloads its transactions with periodic edge connection every 1, 10, 100, 1,000, and 10,000 s.

cost is fundamental to security and would also be required in any other design integrating IoT devices and blockchain.

System Latency. The experiments confirm the feasibility of connecting resource-constrained IoT devices with blockchain technologies. Under fair conditions of around 5-30 min transient connectivity and a fair number of transactions, we show that the memory of small IoT devices is sufficient to store the transactions. For further robustness, we can store them additionally in flash memory. Our result shows that the main bottleneck in terms of memory consumption and end-to-end latency lies on the periodicity of the edge connection.

Security Aspects. We use a fair authentication scheme with a smart contract, which validates each device and transaction. However, each device will need to use a white-list of authenticated nodes, which will increase the local storage needs. We recognize a trade-off between the off-line transaction storage and the number of devices stored in the white-list.

Limitations. A general limitation of our study is the privacy of stored data in a cloud-based blockchain. A blockchain is by nature publicly available for all the stakeholders.

3.7 Related Work

Fair exchange. Similar to our work, FairSwap [79] also proposes the use of smart-contracts for fair exchange, avoiding costly solutions like zero-knowledge proofs. However, the protocol has not been tested and evaluated on any IoT devices.

Blockchains and IoT. Several architectures [38] [39] have been proposed to achieve data provenance using blockchain technology. However, they do not integrate IoT devices, and they do not apply to resource-constrained environments. Nonetheless, there are many benefits [80] of integrating IoT with blockchains. There are approaches to integrate IoT devices with blockchains such as AGasP [40] and Edgechain [41]. The existing architectures are limited in two ways: a) there is a lack of non-repudiation of the transactions created by the IoT nodes, and b) they do not take into account the constraints faced by low-power IoT devices. In contrast, IoTLogBlock focuses on data provenance, achieving non-repudiation, and using low-power IoT devices.

Cryptographic Capabilities on IoT Devices. Previous studies show the affordability of cryptographic operations on small sensor devices [35] [36]. IoTLogBlock differs from previous work in that we use blockchain to record transactions and a contract signing protocol to achieve non-repudiation.

3.8 Conclusion

In this paper, we propose IoTLogBlock, an open-source architecture allowing low-power devices to use a cloud-based ledger to record transactions between devices. We argue that using a blockchain directly is not an option for hardware-constrained IoT devices, as it implicitly assumes the devices are capable of heavy computations and having large memory capacity and always-on communication to the cloud. IoTLogBlock combines an optimistic contract signing protocol, a private-based blockchain, and a smart contract, to cope with these three challenges. The IoT devices create off-line transactions, which later are verified by the smart contract. The blockchain provides to the stakeholders an immutable ledger of all the collected transactions.

We evaluated IoTLogBlock, especially in regards to the extent it allows IoT devices with transient connectivity to create and register transactions in a cloud-based blockchain. We quantify the cost of IoTLogBlock in terms of computation, delay, memory, and energy consumption. It takes on average 3 s for IoT devices to mutual sign and exchange a transaction. Moreover, IoTLogBlock supports intermittent connectivity ranging from 10 s to over 2 h.

3.9 Acknowledgments

This work was supported by the Swedish Research Council (VR) through the project “AgreeOnIT”, the Swedish Civil Contingencies Agency (MSB) through the projects “RICS” and “RIOT”, and the Vinnova-funded project “KIDSAM”.

Paper C

Chistos Profentzas, Magnus Almgren, Olaf Landsiedel

TinyEVM: Off-Chain Smart Contracts on Low-Power IoT Devices

Accepted for publication by the 40th IEEE International Conference on Distributed Computing Systems (ICDCS) 2020

Chapter 4

TinyEVM

With the rise of the Internet of Things (IoT), billions of devices ranging from simple sensors to smart-phones will participate in billions of micropayments. However, current centralized solutions are unable to handle a massive number of micropayments from untrusted devices.

Blockchains are promising technologies suitable for solving some of these challenges. Particularly, permissionless blockchains such as Ethereum and Bitcoin have drawn the attention of the research community. However, the increasingly large-scale deployments of blockchain reveal some of their scalability limitations. Prominent proposals to scale the payment system include off-chain protocols such as payment channels. However, the leading proposals assume powerful nodes with an always-on connection and frequent synchronization. These assumptions require in practice significant communication, memory, and computation capacity, whereas IoT devices face substantial constraints in these areas. Existing approaches also do not capture the logic and process of IoT, where applications need to process locally collected sensor data to allow for full use of IoT micro-payments.

In this paper, we present TinyEVM, a novel system to generate and execute off-chain smart contracts based on sensor data. TinyEVM's goal is to enable IoT devices to perform micro-payments and, at the same time, address the device constraints. We investigate the trade-offs of executing smart contracts on low-power IoT devices using TinyEVM. We test our system with 7,000 publicly verified smart contracts, where TinyEVM achieves to deploy 93% of them without any modification. Finally, we evaluate the execution of off-chain smart contracts in terms of run-time performance, energy, and memory requirements on IoT devices. Notably, we find that low-power devices can deploy a smart contract in 215 ms on average, and they can complete an off-chain payment in 584 ms on average.

4.1 Introduction

As the Internet of Things (IoT) becomes deeply integrated into our daily lives, new opportunities emerge. For example, cities nowadays embed sensors into parking lots to measure occupation. This capability, in turn, allows for new application scenarios, such as smart parking. Once a car approaches an empty parking lot, the lot can automatically inform the car about the hourly parking fees (based on location, time, or other locally set parameters), and engage in their payment when the car drives away. This scenario belongs to a much more generic application setting where the plethora of IoT devices are frequently interacting in two phases. First, they agree on the conditions related to an activity (e.g., the parking fees). Second, at a later time, they are executing/ensuring these conditions have been fulfilled (e.g., payment of the fees). A key challenge here is that the two IoT devices ordinarily do not trust each other, as they are, for example, owned or operated by different entities.

A potential solution to this challenge, worth exploring, are blockchains and their corresponding smart contracts [81, 82]. In principle, via a smart contract stored in a blockchain, a vehicle, and a parking sensor could agree on hourly parking fees, and at a later point in time, enforce the payment (e.g., through micropayments). However, blockchain technologies and smart contracts of today assume powerful nodes that can communicate and synchronize frequently. Ethereum [17], for example, uses a virtual machine to execute smart contracts, where clients need to connect to the blockchain both to upload their transactions and to query for updates.

Contrary, the nodes in IoT networks face constraints in energy, memory, and computation capabilities, making the requirements of current state-of-the-art blockchain technologies an ill fit for the IoT ecosystem. For example, today's resource-constrained devices have some tens of kilobyte memory, which is quickly exceeded by code and state information of smart contracts. High bandwidth, always-on connectivity with 4G or 5G, is infeasible in terms of energy consumption and hardware costs for many applications that shall operate for years on battery power. Moreover, cellular network coverage is far from ubiquitously available. Energy-efficient LPWAN technologies such as LoRa and SigFox, in turn, do not provide the required bandwidth for direct on-line transactions between a smart device and the cloud.

Furthermore, to allow IoT devices to play a central role in future micro-services (e.g., smart parking), they must be able to provide a local context (e.g., sensor data) for the conditions related to the activity about to take place. However, most smart contracts are not well designed to handle input from the outside world. While Oracles [11, 18], as a third-party information source, can supply verified data from Internet-connected sources, there is no direct way for a smart contract to trigger a sensor reading and actuator setting on the IoT sensor-node. Overall, we recognize a gap between high-level blockchain architectures and the need for additional services and the capabilities of IoT devices.

To overcome these challenges, we design TinyEVM, a novel architecture to execute off-chain smart contracts on low-power IoT devices. We begin by revealing the design challenges of an application scenario for payment channels using off-chain smart contracts and introducing three novel approaches. Firstly,

we design the on- and off-chain smart contracts considering the trade-offs between the device constraints and the off-chain protocol. We remove the need for active synchronization of payment channels by using a logical clock. Secondly, we customize the Ethereum Virtual Machine (EVM) to run on resource-constrained IoT devices with just a few kilobytes of memory. Thirdly, we extend the EVM by introducing specific IoT opcodes to allow smart contracts to directly interact with the sensors and actuators of the local IoT device. Our goal is to enable smart contracts written for EVMs to benefit from the large pool of existing contracts and established toolchains for the design, implementation, and verification of smart contracts.

To summarize, our contributions are as follows:

- We design and implement TinyEVM, an open-source[1] system to enable and scale (micro)payments on low-power IoT devices.
- We devise a virtual machine to execute off-chain smart contracts on resource-constrained devices.
- We introduce the concept of specific IoT-opcodes to unify the logic of interacting with sensors and actuators inside a smart contract.
- We quantify the performance of TinyEVM in terms of computation, delay, memory, and energy consumption. Notably, we find that low-power devices (TI-CC2538) can deploy a smart contract in 215 ms on average. The node can complete an off-chain payment in 584 ms on average.
- We finally provide a discussion of implementation challenges and trade-offs regarding off-chain protocols for resource-constrained devices.

Outline. We organize this paper with the following structure. In Section 4.2, we provide the necessary background for TinyEVM. In Section 4.3, we provide an overview of our application scenario and the design requirements. In Section 4.4, we provide the system design of TinyEVM. In Section 4.5, we provide a security analysis of our design. We present the evaluation results in Section 4.6. Finally, we provide the related work in Section 4.7 and the conclusion in Sections 4.8.

4.2 Background

In this section, we provide the essential background to understand the concepts of Blockchains, Smart Contracts, the Ethereum Virtual Machine, Payment Channels (PC), and the Plasma framework.

4.2.1 Overview of Blockchains

A blockchain is a distributed ledger replicated by multiple nodes and kept consistent via a consensus protocol. In cryptocurrencies like Bitcoin and Ethereum, the protocol is called mining. With mining, each node can create a new state by solving a probabilistic mathematical puzzle.

[1] https://github.com/chrpro/TinyEVM

As blockchains became popular, their scalability and performance limitations became apparent [6, 7]. The research community responded with several proposals to scale the blockchains like sharding [9, 10], consensus algorithm variations [1], and trusted execution [11]. In this paper we focus on three prominent proposals: (a) side-chains [12], (b) payment channels [13, 14], and (c) payment networks [15], which we further described below.

4.2.2 Smart Contracts and the Ethereum Virtual Machine

Smart contracts are executable programs stored as bytecode in the blockchain. They highly extend the use of a blockchain as they can programmable change its state. For example, with the so-called Ethereum Virtual Machine (EVM), each node participating in the consensus of the blockchain executes the smart contracts on its local EVM and validate the correctness of the created new state.

The EVM is a Quasi-Turing complete machine [17] to execute state-transitions in the blockchain. The EVM is a 256-bit stack-based machine executing bytecode statements. Each statement consists of an opcode, with 71 active (discrete) opcodes at the time of writing. The machine avoids an infinite execution of the bytecode by charging a fee for each execution statement, the so-called gas. This fee inhibits micro-payments to be affordable, which is why payment channels have been suggested (see below). If a smart contract runs out of gas in the middle of execution, it is aborted.

The current design of EVM treats smart contacts as sequential programs with no support for concurrency. Moreover, EVM does not allow smart contracts to have access to data outside of the network, limiting their usefulness. For a smart contract to include sensor data, current solutions use services such as Oracles [18]. Oracles act as a third-party information source, and supply verified data from Internet sources. TinyEVM proposes a novel approach to include IoT opcodes inside the EVM, where the smart contract can have access to the sensors and actuators of the device.

4.2.3 Payment Channels

The fee(s) for each payment in the blockchain often makes repeated micro-payments unaffordable. Prominent proposals to overcome this limitation are off-chain protocols like payment channels (PC) [13, 14].

With PC, two parties can swiftly exchange small payments and postpone updating the blockchain (avoiding the fees) until they reach a final state. The parties pre-agree and sign a smart contract, which locks a specific amount of money for a specified period. Later, any participant can unlock the funds by providing a final state signed by both participants within the pre-agreed period. As such, the payment channel is a combination of three distinct concepts: 1) **multi-signature addresses**, 2) **time-locks**, and 3) **hash-locks**.

Multi-signature addresses require n-of-n signatures to unlock its funds. The typical case is a 2-of-2 signature address that requires the approval of both parties to unlock the fund. A **time-lock** restricts the validity of the multi-signature to a limited time. A **hash-lock** requires the revealing of the

pre-image of a secret hash value to consider a payment as valid. The payment channel requires at least two on-chain messages to the public blockchain. One message to open the channel and lock the desired amount, including the hash-locks and time-locks. Depending on the design, the channel allows the owner to send messages to update the status or extend the lock-period. With an open channel, two parties perform off-chain payments by exchanging signatures. When they reach the time to close the channel, they reveal the secret-hash.

There are two main extensions of payment channels. First, payment networks [13, 15] reuse existing user channels to route payments off-chain. Second, state channels [45,83,84] provide a general use of channels to store state changes for any application. TinyEVM builds on the design concepts of payment channels and adapts them to the specific requirements of IoT applications.

4.2.4 Side-Chains

Another proposal to scale blockchains is the idea of side-chains [12, 85]. A typical side-chain system includes three main components: 1) an **on-chain smart contract**, 2) **side-chain(s)**, and 3) an **exit function**.

The on-chain smart contract is published in the main-chain, and it acts as a bridge between several side-chains. The smart contract locks the funds in the main-chain that the side-chains can circulate as off-chain tokens. The nodes participating in the side-chain are responsible for maintaining the side-chain. Each side-chain has a mechanism for validating blocks and a fraud-proof mechanism. The fraud-proof(s) is used by the users to report malicious users trying to exit on the main-chain. The exit function allows off-chain nodes to claim the tokens from the side-chain(s). Finally, any node can challenge an exit request on the main-chain using a fraud-proof.

4.3 TinyEVM Overview

This section describes the motivation behind TinyEVM. First, we introduce a simple application scenario and the parties involved therein. Second, we present the system requirements and challenges for low-power devices. Third, we motivate our threat model based on the application scenario.

4.3.1 Application Scenario: Smart Parking

Nowadays, smart parking lots equipped with sensors can detect occupation, and they can interact with a vehicle occupying a spot, for example, via low-power wireless technologies. We envision a marketplace where a car owner and a parking company negotiate the terms of parking and perform micropayments. For this, the car owner and the parking company publish a template smart-contract to a blockchain, which includes the necessary payment information.

When a vehicle approaches the parking lot, and the devices come within range, the lot can initiate the smart contract and create an off-chain payment channel with the vehicle via low-power wireless technologies, see Figure 4.1. The channel includes the common initial deposit of funds by the vehicle. During the parking, they may do multiple transactions and interactions, such as hourly payments or updates on the payment rates based on the time of day. At the

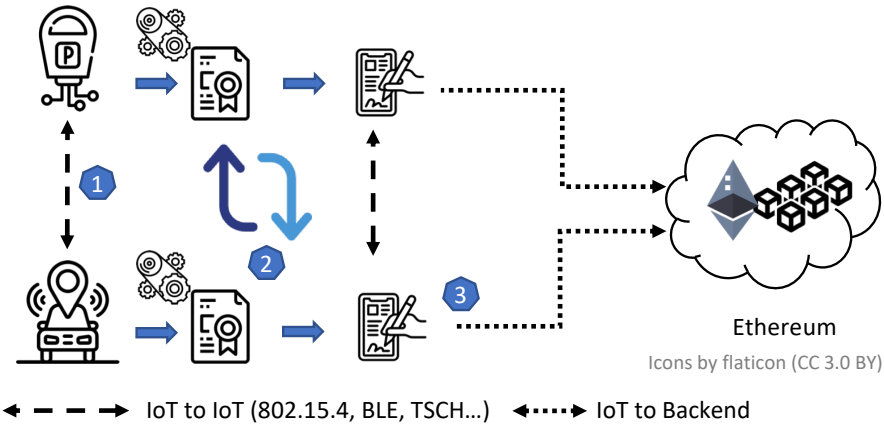


Figure 4.1: The parking application scenario: 1) The vehicle and the parking lot communicate via a short-range protocol. 2) They open an off-chain payments channel with an initial deposit and perform offline payments. 3) A node can at anytime submit a final state to the blockchain, in our case Ethereum.

end of the parking, they close the off-chain channel and sign the final state, which the parking lot can publish to the blockchain to claim the payment.

4.3.2 System Requirements

From the above scenario, we derive the following requirements for the application:

Sensor utilization: The parking lot would like to charge the vehicle owner based on the location of the parking spot, time of day, and possibly other locally relevant conditions, such as the parking availability. Thus, prices can vary and have to be agreed on, for example, via a smart contract.

Low latency: Both parties expect the whole process of negotiating and charging for the parking to be automated and take place in the order of seconds. For this reason, we favor short-range wireless communication for our application scenario.

Low energy consumption: The parking service expects to depend on cheap devices with long battery life-time. The vehicle-owner expects a cheap device easily installed into a regular car.

4.3.3 System Challenges

Beyond the challenge of designing a off-chain protocol, we face other system challenges. To ensure compatibility with off-the-shelf smart contracts and to be able to benefit from the wide variety of tools for smart contract writing, testing, and verification designed by the Ethereum community, one key design goal of TinyEVM is to natively support the Ethereum Virtual Machine (EVM) bytecode. However, EVM is a 256-bit word-size virtual machine. This word-size

leads to two key challenges: (1) resource inefficiency and (2) the complexity of executing 256-bit operations on a 32-bit machine.

First, from a resource perspective, the EVM does not utilize the memory efficiently. All operations are based on 256-bit variables and addresses. As a result, every VM opcode, even a simple addition, operates on 256-bit variables. The main reason for this EVM specification was to ease cryptographic operations like the Keecak256, which works on 256-bit digests. However, the vast majority of variables in a smart contract rarely reach values that require 256-bit storage. Similarly, as our evaluation shows, smart contracts do not reach a size in code or data that they would need 256-bit address space. The result is an inefficient memory use that could prohibit their execution of smart contracts on IoT devices.

The second challenge is the run-time overhead of the virtual machine. The embedded hardware does not directly support 256-bit operations. Instead, we have to emulate 256-bit operations using a 32-bit micro-controller by implementing custom libraries and, as a result, executing a single EVM opcode requires in the order of hundreds of MCU cycles. This inefficiency may further limit the potential of Ethereum smart contracts on resource-constrained IoT devices.

4.3.4 Threat Model

In our scenario, we assume mutually-distrusting, rational parties using a payment channel to exchange payments. The threat model includes two potential threats and focuses on the inability of nodes to revoke previous states of the payment channel. Notably, the node that receives the payment faces the threat of the inability to report a misbehaving peer before the contest period expires. On the other hand, the node sending the payment faces the threat of the inability to unlock her money from the channel.

The receiver expects a **non-repudiation** property of the system. After several payments, none of the parties will be able to deny the participation of the exchange. The parking sensor will be able to claim the money from the blockchain at any time for the time the car has stayed there. On the other hand, the sender expects a **finite** property of the payment channel, which means the channel eventually will close, and the sender can reclaim any remaining money when it leaves.

4.4 TinyEVM System Design

TinyEVM makes four design contributions: First, it separates the transactions of IoT applications into three high-level phases: 1) The on-chain smart contract, 2) the off-chain smart contract, and 3) the on-chain commit(s). Second, TinyEVM customizes the Ethereum Virtual Machine (EVM) to address the resource constraints of IoT devices while staying compatible with the native EVM language. Third, we introduce a template design for smart contracts and the separation of on- and off-chain functionality. Fourth, we extend smart contracts and the EVM with IoT opcodes to interact with onboard IoT sensors and actuators. The new opcodes allow IoT devices to include sensor data and sensor actuation as a part of smart contract development.

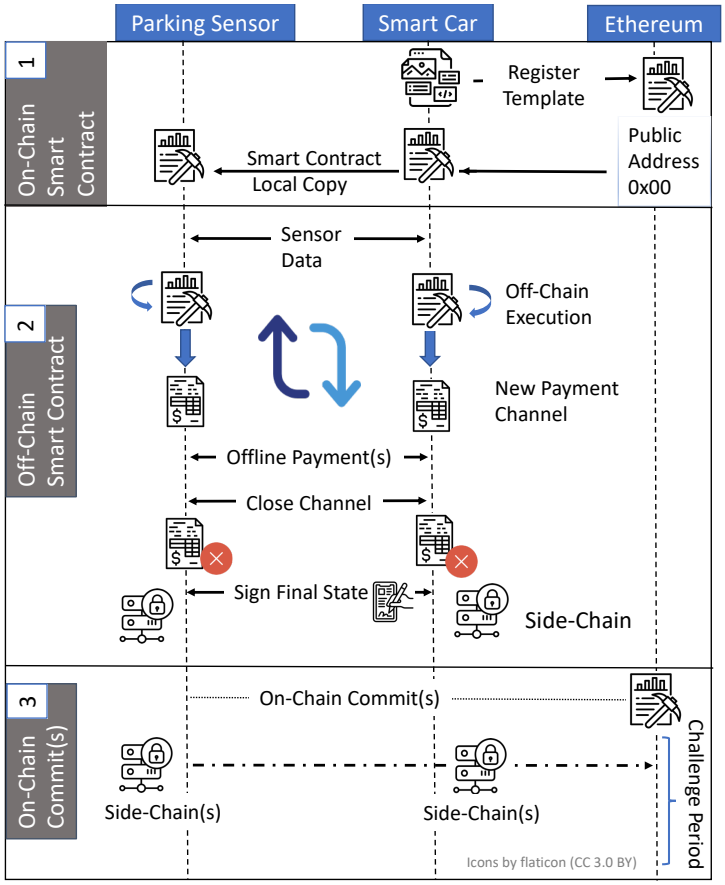


Figure 4.2: The system includes three phases: 1) Publishing the on-chain template smart contract. 2) Creating an off-chain payment channel and signing (multiple) payments. 3) On-chain commit of a final state, which activates the challenge period, and nodes can dispute with their local side-chains.

4.4.1 On- and Off-Chain Transactions: Three Phases

The deployment and execution of smart contracts include three high-level phases, visible in Figure 4.2.

1) On-chain smart contract. A node publishes a smart-contract as a template in the blockchain, which includes the constructor for off-chain payment channels. The node makes a deposit to be charged for parking services, which works as an insurance in case of a dispute. This operation is similar to the on-chain contract used in side-chains. The on-chain smart contract serves as a bridge between the blockchain and our off-chain payment channels.

2) Off-chain smart contract. The nodes use the template to deploy a new off-chain payment channel using a unique monotonic counter (logical clock) as an identifier. The off-chain payment channels are locally generated smart contracts that allow the use of sensor data. Nodes may use the sensor readings, actuator interactions, or data received from other IoT devices as a part of

Component	EVM	TinyEVM
Stack memory	256-bit	256-bit
Random access memory	8-bit	8-bit
Storage space	256-bit	8-bit
Operation opcodes	27	27
Smart contract opcodes	25	21
Memory opcodes	13	13
Blockchain opcodes	6	-
IoT opcodes	-	1

Table 4.1: Comparison of the original EVM and the TinyEVM specifications. The word size of the stack and random access memory remains the same. TinyEVM removes the opcodes related to (on-)blockchain operation, it uses 8-bit side-chain storage space, and it introduces new IoT-specific opcodes.

the smart contract. The sensor data can range from weather conditions or device location, combined with the knowledge about the occupation of nearby parking spots – derived, for example, from the LIDAR data of a modern car – which can be used to evaluate and negotiate the parking fees. The nodes continue with the signing and exchanging of off-line payments until they close the payment channel. The nodes can open and close an arbitrary number of payment channels, but limited to the money deposited and locked in the on-chain smart contract.

3) On-chain commit. At any time, a node can exit from an off-chain channel by publishing a final channel state or a (side-chain) log of its local execution. Our commit function checks the logical clock of the channel and the validity of the signatures. In the case of a correct state, the new state is appended to the tree of the on-chain smart contract. The other node can challenge the state using the local log(s) of the off-chain payments. Finally, there is an exit function that a node can activate, which stops further updates from off-chain channels. The activation of the exit function starts the expiration period, and then it will dissolve the on-chain smart contract and return any unspent money. During that time, the other node can dispute the latest state and claim the insurance money, as common for established side-chains.

4.4.2 Customized Ethereum Virtual Machine

We enable smart contracts on IoT devices by customizing the Ethereum Virtual Machine (EVM) to meet the constraints of IoT devices, especially in terms of device memory. In Table 4.1, we list the specifications of the original EVM compared with our customized one. There are three types of memory that a smart contract can use: 1) stack memory, 2) random-access memory, and 3) storage memory. We achieve to meet the memory requirements without the loss of functionality and compatibility of the IoT device.

The original EVM defines a 256-bit word machine, which is not directly supported on a 16-bit or 32-bit micro-controller. However, we keep the same word-size for compatibility reasons, and emulate a 256-bit word-size in our

implementation. This implementation allows us to use the original Ethereum bytecode with no modifications. However, the storage space is irrelevant for off-chain executions. These operations are needed only for the main-chain. For the off-chain computations, we utilize an 8-bit storage space to store only the side-chain created by the IoT nodes.

We list the machine opcodes into five categories. First, the operation opcodes define the necessary computations, like addition and multiplication. The original EVM supports 27 operations, and TinyEVM supports all of them. Second, the smart contract opcodes are related to smart contract execution like method calls, and returns. TinyEVM supports the necessary operations except for the GAS operations. There is no charging for the off-chain computations as all operations are executed locally. Third, the memory opcodes are related to operations on memory like store and load, and TinyEVM supports all of them. Fourth, the blockchain opcodes are used to get information from the blocks of the blockchain. TinyEVM does not support any block-related opcode since there is no access to the blockchain during local execution. Finally, TinyEVM introduces a novel opcode for IoT sensor data. This extension allows us to include sensor data inside the smart contract.

We observed that the original EVM includes unused opcode(s) that are not currently in use. Thus, we utilized one of the unused opcodes to introduce the IoT sensor functionality. In detail, we use the 0x0c undefined opcode to represent the action of sensing or actuating on the device. Details, such as which sensor to use and additional parameters are given as options to the opcode. This allows us to include arbitrary types of sensor data, and the TinyEVM hides the implementation details from the user.

4.4.3 On-Chain Smart Contract

For our purposes, we assume that the entity providing a service (e.g., the parking service) has published the parking conditions as a smart contract template on the blockchain. The template includes all rules that the two parties need to create and use an off-chain payment channel. Upon accepting the conditions, the user (e.g., the owner of the car) locks the desired amount to be used for the services. Alternatively, if both parties have to negotiate some details, an additional negotiation phase is possible to construct the template [86].

The template is a factory-smart-contract [87], which can create and deploy child contracts dynamically, see Listing 4.1. The child contracts in our scenario are the payment channels, see Listing 4.2.

4.4.4 Off-Chain Smart Contract

The second phase in Figure 4.2 starts when both entities come within range of their low-power wireless technologies, e.g., a smart-car and smart-parking lot come within range. The nodes exchange their sensor data and transactions via a short-range protocol like TSCH [74] or BLE [88]. Please note that the design of TinyEVM is agnostic to the specific technology used. Both entities execute the bytecode of the template to generate an off-chain payment channel. The payment channel includes an ID and the sequence number, which uniquely identifies each transaction on the channel. The sequence number acts as a

```
contract Template {
  address[] PaymentChannels;
  uint Balance;
  address payable public Receiver;
  uint Logical-Clock = 0;
  MerkleSumTree Side-Chain-Root;

  function CreatePaymentChannel (uint64 Money) public  {
      newPaymentChannel = new PaymentChannel( receiver, Money);
      PaymentChannels.push(newPaymentChannel);
      Logical-Clock += 1;
  }
  function OnChainCommit{...} //user specific
  function Challenge{...} //user specific
}
```

Listing 4.1 : Factory Template in Solidity

```
contract PaymentChannel {
 address payable public Sender;
 address payable public Receiver;
 uint public sensor_data;

constructor(address payable _recipient) public payable {
  sender = msg.sender;
  recipient = _recipient;
  assembly{
      0x0c //IoT sensor opcode
      sstore(0x0c) // Store sensor data
  }
}
function close(uint amount, bytes memory signature) public
    payable {
  require(msg.sender == recipient);
  require(isValidSignature(amount, signature));
  recipient.transfer(amount);
  selfdestruct(sender);
}
}
```

Listing 4.2 : Payment Channel in Solidity

logical clock, which is different from the real-time bound of the original concept of payment channels. By using sequence numbers, we can determine the causal order of the payments. With the casual order, the channels can capture the order in which the payments happens, but not the actual time that they occur. The use of logical clocks loosens the requirements for communication and synchronization.

Each device maintains a sequence number that uniquely identifies each of its transactions by simply incrementing a counter for each new transaction. The sequence number is later used for verification and ensures that no device skips reporting any transactions. With the off-chain payment channel, the two nodes can perform several off-chain payments by exchanging signed transactions (using their low-power wireless radios). The signed off-chain payments are stand-alone artifacts that can claim money from the main-chain. The signed payment includes information on the payment channel ID and its unique counter, which makes it trivial to verify the logical order. A node can report either the payment or the final state of the channel, which aggregates all other previous payments. Each execution of the payment channel extends the local (side-chain) log of the node, which links each state with the previous. The local (side-chain) log uses the root published on the main-chain smart contract, which allows verification of the logical order of the executions and ensures that no transactions are omitted. The nodes use the off-chain payment channel repetitively to perform several payments until they close it. The total amount of payments is limited by the funds locked in the main-chain.

4.4.5 On-Chain Commit & Challenge Period

At any time, a node can submit a signed final state of a closed off-chain payment channel. The sequence number of the channel allows the nodes to retrieve the money asynchronously. The node can submit a state that happened after the latest submit without providing the details of the actual time that it happened. Every time the on-chain contract receives a request, it verifies the signed states and updates its sequence number to the highest that received.

The on-chain smart contract uses a Merkle-Sum-Tree [12], which has the sum of the payments and the hash value. The sum value is used as a validation condition along with the hash value. This condition makes it possible for auditing the sum of the payments. Each payment adds to the overall sum, and if it exceeds the allowed range, the payment is invalid, and the other node can claim the insurance money. In our system, we further extend the validation condition with the sequence numbers. Reporting a state with a higher sequence number accumulates the changes of the previous states.

Finally, the sensor node can activate the exit function by submitting a signed final state. This action restricts further submission and starts the challenge period. The other node can submit a transaction with a higher sequence number value to claim the insurance money. As each transaction is signed, it is not disputable.

4.5 Security Analysis

Similar to other off-chain systems [12, 46], TinyEVM does not entirely prevent double-spending fraud but instead makes it unprofitable. We design our system based on three security properties. 1) We can detect any fraud using the sequence numbers and the signatures of the participants. 2) We introduce proper punishment and incentives for the participants to report any misbehavior. 3) We have a time-limit in which a party can claim the money deposited in the on-chain contract.

Detection: Each time a node performs a transaction or closes a channel, it increases the sequence number, and it eventually will report the local state to the blockchain. The on-chain smart contract always stores the most recent state, i.e., the one with the highest sequence number. As a result, the sequence number prevents a node from misbehaving by reporting old states. Reporting a signed transaction or state with a higher sequence number denotes a valid next state.

A node could exchange a transaction with another node, and it may skip to report or even try to delete the transaction. If the other node behaves correctly, it will upload all the transactions, and the on-chain smart contract detects the misbehaving node. Thus, a transaction will only go unnoticed if both involved parties are misbehaving. We argue that it is very uncommon for two parties to conspire in this way, as they commonly do not share the same goal.

Non-repudiation: One party could claim the money from the blockchain at any time by providing a signed state. On the other hand, the other party can close the channel to refund any unspent money at any time. The system is based on incentives that penalize misbehavior. The first party has the incentive not to overspend the locked funds; otherwise, the insurance money will be lost. The second party has the incentive to report the last payment before the template expires.

Time-limit: The template has an exit function that allows the sensor owner to start the process to renounce any unspent money. This time-limit is in order of days similar to the popular framework (e.g., Plasma, which has a seven-day bound) [12].

4.6 Performance Evaluation

In this section, we present the evaluation of TinyEVM on low-power IoT devices. The evaluation responds to the following questions. (a) Is TinyEVM technically feasible on low-power IoT devices? (b) What is the performance of executing a smart contract on a low-power device? (c) What is the overhead in terms of computation, memory, and energy consumption of the off-chain functionality, including both wireless communication and local processing.

Outline. First, we list the implementation details and target platforms. Second, we present the evaluation of the customized Ethereum Virtual Machine (EVM) on low-power devices. This part of the evaluation represents a macro-benchmark of the system regarding the ability to deploy smart contracts on low-power devices. Third, we present the evaluation of the off-chain functionality in terms of run-time performance, energy, and memory requirements for both communication and local computation. The off-chain evaluation represents

a micro-benchmark of the system, and it provides details on executing the off-chain payment-channel application.

4.6.1 Experimental Setup

Implementation. We implement the Ethereum Virtual Machine (EVM) in C as a module for the Contiki-NG OS. For wireless communication, we use the TSCH protocol stack provided by Contiki-NG. We support smart contract deployment up to 8 KB of bytecode. We implement EVM as a 256-bit word size machine with 3 KB of stack, 8 KB of random access memory, and 1 KB for off-chain storage. A comparison between the original specifications and TinyEVM is presented in Table 4.1.

Hardware Setup. Building on cryptographic primitives, our implementation targets sensor nodes with cryptographic hardware support. We use Openmote B that is based on the TI-CC2538 SoC [47]. The SoC runs a 32-bit ARM Cortex-M3 CPU at 32 MHz, 32 KB of RAM, and 512 KB of ROM. Moreover, the SoC features a cryptographic engine at 250 MHz and an 802.15.4 radio transceiver. Please note that TinyEVM is not bound to this particular platform, and it can be deployed on any platform supported by Contiki-NG, as long as it has a cryptographic co-processor, sufficient resources, and a 802.15.4 radio interface.

4.6.2 Ethereum Virtual Machine on IoT Devices

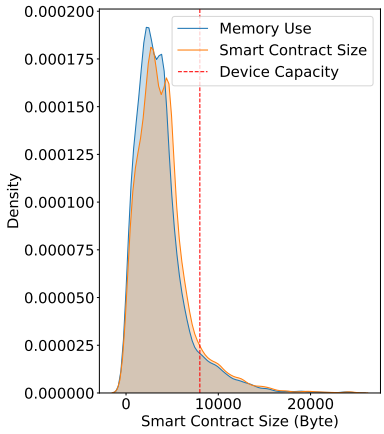
We evaluate the resource efficiency of TinyEVM and its ability to deploy off-the-shelf smart contracts. For this, we collect roughly 7,000 publicly available smart contracts to test our platform. The smart contracts are verified by Etherscan.io, a widely used blockchain explorer.

4.6.2.1 Memory Requirements

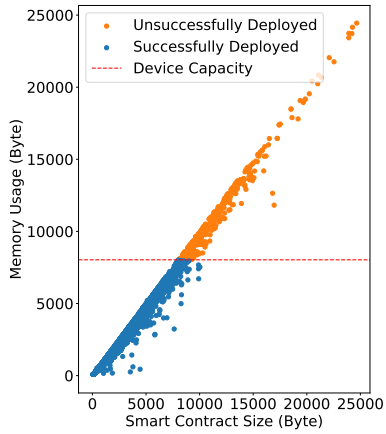
The deployment of a smart contract starts with the initialization of the smart contract using its constructor function. This function initializes all the variables, and it will take the initial steps to make the smart contract executable. Finally, it will return the actual bytecode that will be installed on the device.

In Figure 4.3a, we see the distribution of the smart contract size and the memory usage in TinyEVM after the deployment. We observe that the average size of a smart contract is 4 KB (see also Table 4.2). The maximum size is 25 KB and the minimum size is 28 Bytes. The sizes of the smart contracts are within the capacity of the device that facilitates 32 KB of RAM. The deployment limit is set to 8 KB (red-dotted line); If we want to deploy larger smart contracts, we need to allocate less memory for the system configuration (see Table 4.3, e.g., stack size). Such allocation will lead to execution failures, e.g., stack overflows, and we argue that 8 KB represents a favourable memory allocation point.

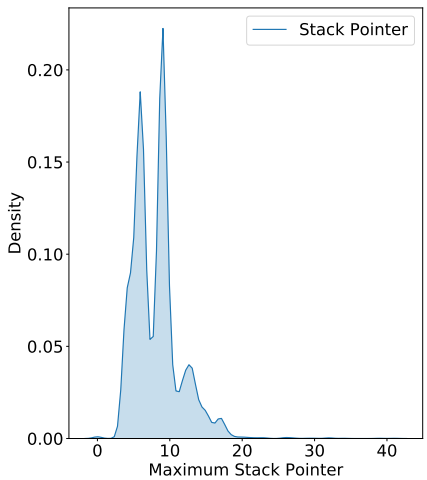
In our experiment, we successfully deploy 93% (5,953) of the public smart contracts on the low-power device without any modification. All other contracts fail due to resource limitations. In Figure 4.3b, we observe the memory usage needed to deploy the smart contracts. We notice the positive correlation between memory use and the size of the smart contract. However, the final



(a) The distribution of the memory requirements for 7,000 smart contracts. We are able to deploy 93% (5,953) of the smart contracts on the low-power device. The memory capacity is set to 8 KB. For the remaining 7% we would need more memory on the device.



(b) Device memory usage in relation to the smart contract size. There is a positive correlation between the smart contract size the device memory requirements. The memory required for the deployment is never longer than the size of the contract.



(c) The use of the stack memory of the successful deployed smart contracts. The figure shows the maximum value reached by the stack pointer.

Figure 4.3: The graphs presenting the memory usage of the smart contract deployment. This includes the density distribution regarding the memory and stack usage of the virtual machine.

deployment never requires more memory than the actual size of the smart contract. This behavior allows the device to deploy some outliers with bytecode size higher than 8 KB, but with a final deployment requirement of less than 8 KB.

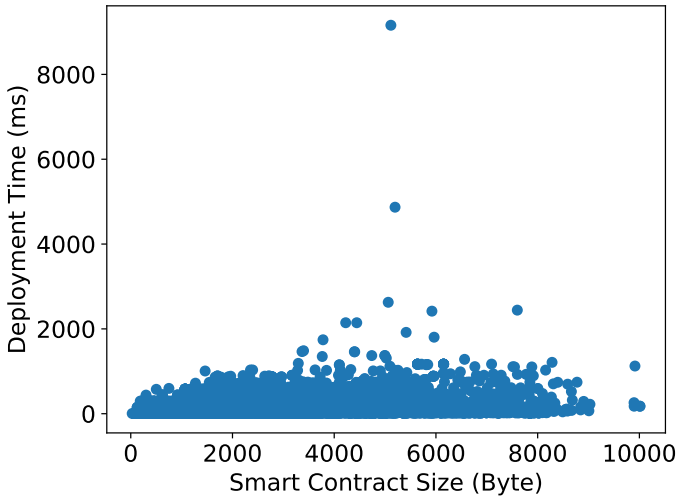


Figure 4.4: The time of deploying a smart contract in relation to its bytecode size. The average time is 215 ms, but we can notice there are some outliers.

We continue our analysis with the virtual machine's stack usage during the experiments. In Figure 4.3c, we present the distribution of the maximum Stack Pointer (SP) during execution. We can observe the majority of the smart contracts use a maximum of ten elements. In Table 4.2, we observe that the maximum SP is 41 elements, with an average of 8 elements. As a reminder, the Ethereum specifies a maximum SP of 1024 elements. From our experiment, we see a tendency of developers to write small, concise smart contracts, which rarely need more stack during execution. However, the execution of deployed smart-contract functions can vary depending on the parameters. The evaluation of these functions is hard to define in practice as their parameters are not known before the actual execution. However, our evaluation gives some insights regarding the design choices for the virtual machine. Overall, we conclude that in terms of random access memory, stack memory, and storage memory of TinyEVM, we support the majority of the 7,000 publicly available smart contracts.

4.6.2.2 Deployment Execution Time

In Figure 4.4, we present the deployment time (in ms) compared to the size of the bytecode. As a first observation, there is no correlation between the size of the bytecode and the deployment time. However, we only perform the deployment of the smart contract in our evaluation with the default parameters.

We see in Table 4.2 that the average execution time is 215 ms, with a standard deviation of 277. It is worth to notice that we observe some outliers that need more time to deploy the smart contract, which highly depends on the nature of the opcodes they use. The maximum time we observe is 9.2 seconds, showing that smart contracts can be deployed within seconds, even on resource-constrained IoT devices.

4.6.3 Off-chain Payment Channels

Next, we give insights into the memory, CPU performance, and energy consumption of the execution of off-chain payment channels on low-power devices. We run our experiments over 200 times, and we report the standard deviation when it is not negligible (as σ). For the energy consumption of Contiki-NG, we rely on the internal Energest module [89] that has a 30-microsecond resolution timer.

4.6.3.1 Memory Requirements.

We present the memory foot-print of the payment channel in Table 4.3 and divide it into three main parts: (a) The Contiki-NG is the operating system including the necessary network stack and libraries, (b) the smart contract template to generate payment channels, and (c) the TinyEVM is our customized Ethereum Virtual Machine (EVM).

Contiki-NG itself consumes 33% of the available RAM. This module is necessary for the general functionality of the device and also provides the wireless protocol stack. The EVM has a significant impact and consumes 42% of the RAM. The smart contract template, which we implemented for this evaluation scenario, is deployed as bytecode and consumes only 5% of the RAM. Finally, the whole program consumes only 11% of the ROM. The deployed smart contract fully supports our smart parking application scenario, and the results underline that we have resources for significantly more complex application scenarios.

4.6.3.2 Cryptographic modules

Next, we evaluate the performance of the cryptographic functions. The keccak256 hash function is based on software implementation, as the cryptographic hardware of our platform does not support it. All the other cryptographic operations are performed by the cryptographic hardware running at 250 MHz. In Table 4.5, we present the performance of each separate task. The average time to complete all cryptographic functions of a complete transaction round is 356 ms. The most time-consuming operation is the ECDSA signature, which takes 350 ms. We argue that this overhead, while significant, is feasible for a large number of applications. For example, in the car parking application scenario, it means a user will need less than a second to sign and exchange a valid transaction.

4.6.3.3 Energy consumption

We present the energy consumption of off-chain transactions on resource-constrained IoT devices. Focusing on the energy consumption of the off-chain payment channel, we report our results after the TSCH node discovery. Node discovery happens quickly [74], and the energy consumption is insignificant. Moreover, this discovery is specific to the TSCH protocol and would have a different footprint on other communication technologies such as BLE. In Figure 4.5, we depict the flow of the electric current (in mA) for a full-round of the off-chain process. The process involves three discrete pieces: wireless communication, the virtual machine execution, and the cryptographic engine.

Measurement	Contract Size	Stack Pointer	Stack (Bytes)	Memory (Bytes)	Deployment Time (ms)
Max	10,058	41	3,056	8,056	9,159
Min	28	3	768	96	5
Mean	4,023	8	2,048	3,676	215
Std	2899	3	827	2,801	277

Table 4.2: An overview of memory and deployment time of the 5,953 successfully deployed smart contracts.

Component	RAM Bytes	RAM Percent	ROM Bytes	ROM Percent
Contiki-NG OS	10,394	33%	40,527	10%
TinyEVM	13,286	42%	1,937	1%
Smart Contract Template	2,035	5%	-	-
Total footprint	25,715	80%	53,239	11%
Available memory	6,285	20%	458,761	89%

Table 4.3: Memory Footprint of TinyEVM (max sizes) on CC2538, which facilitates 32KB of RAM and 512 KB of ROM.

State	Time [ms]	Current [mA]	Energy [mJ]
Cryptographic Engine	350	26	19.1
TX	32	24	1.6
RX	52	20	2.1
CPU @ 32 MHz	150	13	4.1
CPU @ LPM2	982	1.3	2.7
Total	1,566	-	29.6

Table 4.4: Derived energy consumption of running the contract signing protocol on the CC2538 SoC given a supply voltage of 2.1 V. Note we configure Contiki-NG to use the low-power mode 2 (LPM2) [47], when not active.

As a first step, in the parking scenario, the nodes exchange their data. Our evaluation includes reading sensor data, in this case from the temperature sensor, to evaluate the overhead of IoT sensor and actuator operations. The smart car starts by sending its sensor data at 0.25 s, followed by the receiving of parking sensor data visible in Figure 4.5. Second, the car at 0.45 s executes the smart contract to create the off-chain payment channel. This execution takes on average 0.20 s with a σ of 0.1.

Third, the car signs a payment for the parking sensor, where the signature takes 0.35 s on average. In practice, such payment would be conducted at an application-specific rate, commonly in the order of minutes. For brevity, we include only one payment here. At the end of the parking, the car executes the off-chain payment channel to register the payment on the side-chain. This process takes on average 0.08 s with a σ of 0.01. Finally, the car exchanges signatures with the parking sensor.

In Table 4.4, we report the total energy consumption (in mJ) of the off-chain process. We notice that the major energy consumption (65%) comes from the cryptographic engine with 19.1 mJ. The wireless communication using the TSCH protocol contributes to 3,7 mJ (13%). Finally, the execution of the virtual machine contributes to a CPU consumption of 4,1 mJ (14%). In total, the off-chain process consumes 29,6 mJ. We observe that the cryptographic engine is the main energy consumer, while both the virtual machine and wireless communication consume considerably less.

By default, the OpenMote platform is powered by two standard AA alkaline

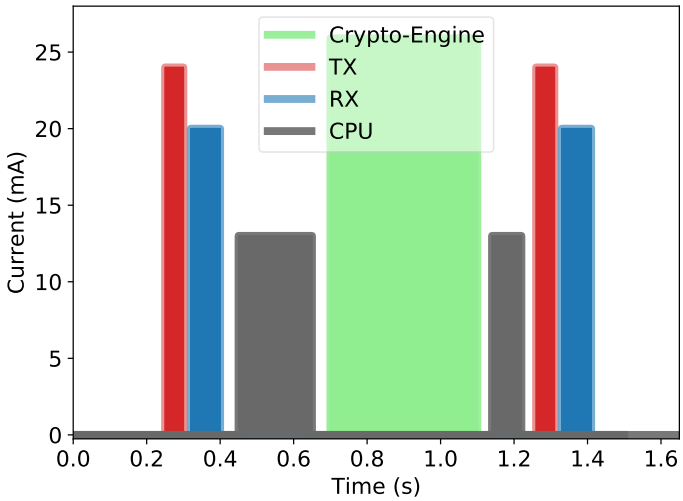


Figure 4.5: The electric current (in mA) drawn by a complete round of the off-chain payments channel. There are three discrete pieces involved: the wireless communication, the virtual machine execution, and the cryptographic engine.

Operation type	**Mode**	**Time**
ECDSA - Signature	HW	350 ms
SHA256 - Hash function	HW	1 ms
Keccak256 - Hash function	SW	5 ms
Total time		356 ms

Table 4.5: Performance of cryptographic operations. Keccak256 is implement on software, the other operations are using the CC2538 cryptographic engine running at 250 MHz.

battery of 2500 mAh. Thus, we can expect 10,000 Joules of energy from the cells, which allows us to perform roughly 333,000 payments. If we assume one payment on average every 10 minutes, this would lead to a battery life-time of more than six years. While this is merely an estimate and other factors such as energy consumption during deep sleep and battery leakage need to be considered, we argue that this order of magnitude of payments is practical for a wide range of application scenarios, including our parking example.

4.7 Related Work

We list the related work in five parts: (1) blockchains and IoT, (2) scaling of blockchains using off-chain protocols including payment channels and networks, (3) payment hubs, (4) side-chains and oracles, and (5) virtual machines in the context of IoT and wireless sensor networks.

Blockchain and IoT. Previous proposals highlight the benefits of using a blockchain for IoT applications. IoTLogBlock [82] demonstrates the feasibility of creating and validating transactions using low-power devices for cloud-based blockchains like Hyperledger. However, IoTLogBlock is not applicable to public blockchains such as Ethereum and Bitcoin. AGasP [81] analyses the benefits of using smart contracts for IoT applications. This architecture assumes powerful nodes able to interact and synchronize with the main blockchain. TinyEVM proposes the off-chain execution of smart contracts and involves sensor data as part of the execution.

Payment Channels & Networks. Researchers have proposed several improvements and extensions to Payment Channels (PC). A major extension to PC is the payment networks, where users can reuse existing PCs to form a routing network. On the commercial side, the Lighting Network [90] was one of the first implementations of such networks for Bitcoin. The equivalent network for Ethereum is Raiden [91].

There are three main challenges to make PCs usable in practice. One challenge is to open a PC in both directions. Duplex micropayments [92] are an extension to allow the user to have this type of PC. Second, a user cannot reallocate the locked money in the channel. Revive [13] allows a user to rebalance the payment channels and to reallocate money to a channel without the cost of closing and reopening it. Third, there are privacy concerns regarding the ability of track payments. Bold [14] tackles privacy issues and ensures that multiple payments are unlinkable with the assumption that the participants use anonymized capital. Other proposals solve the privacy issues with different trade-offs, for example SilentWhispers [44], and SpeedyMurmurs [93]. However, the above approaches assume active communication and synchronization among nodes, which is not a valid assumption in the context of IoT, especially resource-constrained IoT.

Payment Hubs. The idea of a payment hub is to use the nodes that have multiple open channels (hub) to circulate the payments. The other nodes need to connect to a hub node. There are three challenges for the payment hubs.

First, there are concerns about the anonymity of the payments. Tumblebit [84] makes the payments unlinkable by using an untrusted intermediary. Second, there is the involvement of the intermediary for each payment. This intermediary leads to performance issues. Perun [94] proposes the virtual payment hub to avoid this problem. Third, payment hubs can lead to collateral fragmentation. NOCUST [95] proposes the separation of the functionality of the payment hub to two components. First, an off-chain operator server handles every transfer. Second, an on-chain smart contract verifies the payments. Finally, Ye et al. [15] propose a system (Boros) to shorten the payment path for the hub network. In the context of IoT, a payment hub allows us to scale the payment system, but several trade-offs need to be evaluated. This is one focus of our future research.

Side-Chains. This proposal [46] includes an on-chain smart contract acting as a bridge between several lighter and faster side-chains. The nodes can exchange the off-chain tokens that have a correspondence in the main blockchain.

The Plasma [12] framework is the proposal of the Ethereum team to scale the blockchain network. However, the current side-chains do not take into

consideration the challenges of low-power IoT devices. Our system is built on top of these ideas, and we further extend the functionality of off-chain smart contracts to have access to sensors and actuators of the IoT device.

Oracles. An oracle provides a solution to the design inability of Ethereum to include data from the physical world (e.g., sensor data). TownCrier [11] provides a bridge between HTTPS-enabled data websites and the Ethereum blockchain. Moudoud et al. [18] show a test case of an IoT supply chain scenario using a network of oracles and smart contracts. Our system differs from these proposals since TinyEVM proposes a novel approach where the smart contract can have access directly to the sensors and actuators of the IoT device.

Virtual Machine for IoT. Several virtual-machines [96–98] have been proposed for IoT devices before to provide support for high-level languages such as Java. However, most of them define word-size from 8-bit to 32-bit indexes, which are natively supported. In TinyEVM, we design a 256-bit machine tailored for off-chain smart contracts, which brings new challenges. A limitation of the Ethereum Virtual Machine (EVM) is its limited support for concurrency, which TinyEVM inherits. One suggested solution for concurrent execution is to use speculatively parallel executions of smart contracts [99], however, these solutions are not designed for resource-constrained devices.

4.8 Conclusion

In this paper, we present TinyEVM, a novel system to perform off-chain payments using low-power IoT devices. TinyEVM allows deploying smart contracts from powerful nodes on a resource-constrained device. We also extend the functionality of the smart contracts to have access to sensor reading and actuation of the device as part of the high-level code, allowing the integration of could-services with IoT-nodes.

TinyEVM achieves a sweet spot between the scalability requirements of the blockchain and the off-chain computation. The design of TinyEVM focuses on the energy and memory requirements of low-power devices. Our evaluation shows the technical feasibility of executing off-chain smart contracts on IoT devices. We deploy 5,953 smart contracts with an average of 4 KB size with the average deployment time of 215 ms. Finally, we evaluate the execution of off-chain smart contracts in terms of run-time performance, energy, and memory requirements on IoT devices. Notably, we find that low-power devices can deploy a smart contract in 215 ms on average. The IoT node can complete an off-chain payment in 584 ms on average.

As future work, we will investigate the feasibility of payment networks and payment routing algorithms on low-power IoT devices. Also, we will improve some of the privacy concerns of off-chain payments.

4.9 Acknowledgments

This work was supported by the Swedish Research Council (VR) through the project “AgreeOnIT”, the Swedish Civil Contingencies Agency (MSB) through the projects “RICS” and “RIOT”, and the Vinnova-funded project “KIDSAM”.

Bibliography

[1] L. M. Bach, B. Mihaljevic, and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[3] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol," in *Advances in Cryptology*, J. Katz and H. Shacham, Eds. Springer International Publishing, 2017.

[4] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, 2002.

[5] K. Croman, C. Decker, I. Eyal, and A. E. Gencer, "On scaling decentralized blockchains," in *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2016.

[6] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[7] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *ACM Conference on Management of Data (SIGMOD)*, 2017.

[8] T. T. A. Dinh *et al.*, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *ACM Conference on Management of Data (SIGMOD)*, 2017.

[9] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding," *IEEE Symposium on Security and Privacy (SP)*, 2018.

[10] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[11] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[12] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *Plasma.io*, 2017.

[13] R. Khalil and A. Gervais, "Revive: Rebalancing Off-Blockchain Payment Networks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[14] M. Green and I. Miers, "Bolt: Anonymous Payment Channels for Decentralized Currencies," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[15] Y. Ye, J. Zhang, W. Wu, X. Luo, and J. Cao, "Boros: Secure Cross-Channel Transfers via Channel Hub," *arXiv*, 2019.

[16] J. Eberhardt and S. Tai, "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations," in *IEEE Conference on Internet of Things (iThings) and Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018.

[17] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger EIP-150," *Ethereum Yellow Papers*, 2017.

[18] H. Moudoud, S. Cherkaoui, and L. Khoukhi, "An IoT Blockchain Architecture Using Oracles and Smart Contracts: the Use-Case of a Food Supply Chain," in *IEEE Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2019.

[19] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 3rd ed. Pearson Education, 2002.

[20] J. D. Tygar and B. S. Yee, "Dyad: A system for using physically secure coprocessors," *Technical Report CMU-CS-91-140R*, 1991.

[21] Jianying Zhou and D. Gollman, "A fair non-repudiation protocol," in *IEEE Symposium on Security and Privacy*, 1996.

[22] D. Boneh and M. Naor, "Timed commitments," in *Advances in Cryptology (CRYPTO)*. Springer, 2000.

[23] N. Asokan, V. Shoup, and M. Waidner, "Asynchronous protocols for optimistic fair exchange," in *IEEE Symposium on Security and Privacy*, 1998.

[24] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[25] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings. 1997 IEEE Symposium on Security and Privacy*, 1997.

[26] O. Khalid, C. Rolfes, and A. Ibing, "On implementing trusted boot for embedded systems," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2013, pp. 75–80.

[27] I. Lebedev, K. Hogan, and S. Devadas, "Invited Paper: Secure Boot and Remote Attestation in the Sanctum Processor," in *IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018.

[28] The Chromium OS team, "Verified boot," http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot, 2009.

[29] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, 2018.

[30] M. Hossain and R. Hasan, "Boot-iot: A privacy-aware authentication scheme for secure bootstrapping of iot nodes," in *2017 IEEE International Congress on Internet of Things (ICIOT)*, June 2017, pp. 1–8.

[31] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation." in *NDSS*, 2013.

[32] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 268–286.

[33] N. Asokan, V. Shoup, and M. Waidner, "Optimistic fair exchange of digital signatures," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 4, 2000.

[34] H. Shafagh, A. Hithnawi, A. Droescher, S. Duquennoy, and W. Hu, "Talos: Encrypted query processing for the internet of things," in *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[35] A. L. M. Neto, H. K. Patil, L. B. Oliveira, A. L. F. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. F. Loureiro, and D. F. Aranha, "AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle," in *ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2016.

[36] A. Liu and P. Ning, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," in *IEEE Conference on Information Processing in Sensor Networks (IPSN)*, 2008.

[37] O. Novo, "Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT," *IEEE Internet of Things Journal*, vol. 5, no. 2, 2018.

[38] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "ProvChain: A Blockchain-Based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability," in *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[39] R. Neisse, G. Steri, and I. Nai-Fovino, "A blockchain-based approach for data accountability and provenance tracking," in *ACM Conference on Availability, Reliability and Security (ARES)*, 2017.

[40] Y. Hanada, L. Hsiao, and P. Levis, "Smart Contracts for Machine-to-Machine Communication," in *IEEE Conference on Internet of Things and Intelligence System (IOTAIS)*, 2018.

[41] J. Pan, J. Wang, A. Hester, I. Alqerm, Y. Liu, and Y. Zhao, "EdgeChain: An Edge-IoT Framework and Prototype Based on Blockchain and Smart Contracts," *arXiv:1806.06185 [cs]*, 2018.

[42] E. Androulaki, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, A. Barger, S. W. Cocco, J. Yellick, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, and G. Laventman, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *ACM European Conference on Computer Systems (EuroSys)*, 2018.

[43] S. F. Aghili, M. Ashouri-Talouki, and H. Mala, "DoS, impersonation and de-synchronization attacks against an ultra-lightweight RFID mutual authentication protocol for IoT," *The Journal of Supercomputing, Springer*, 2018.

[44] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei, "SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks," *Network and Distributed System Security Symposium (NDSS)*, 2017.

[45] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," *arXiv*, 2017.

[46] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "SoK: Off The Chain Transactions," *IACR Cryptology ePrint Archive*, 2019.

[47] Texas Instruments, "CC2538 System-on-Chip for 2.4-GHz IEEE 802.15.4," www.ti.com.cn/cn/lit/ug/swru319c/swru319c.pdf, 2013.

[48] M. Gerla, E. K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 241–246.

[49] Z. Bi, L. D. Xu, and C. Wang, "Internet of things for enterprise systems of modern manufacturing," *IEEE Transactions on Industrial Informatics*, 2014.

[50] S. Kashyap, V. S. Rao, R. V. Prasad, and T. Staring, "Cook over ip: Adapting tcp for cordless kitchen appliances," in *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, April 2018, pp. 1–12.

[51] S. Berger, K. Goldman, D. Pendarakis, D. Safford, E. Valdez, and M. Zohar, "Scalable attestation: A step toward secure and trusted clouds," *IEEE Cloud Computing*, vol. 2, no. 5, pp. 10–18, Sept 2015.

[52] The Android Team, "Verifying boot," https://source.android.com/security/verifiedboot/verified-boot, 2017.

[53] S. Eresheim, R. Luh, and S. Schrittwieser, "On the impact of kernel code vulnerabilities in iot devices," in *International Conference on Software Security and Assurance (ICSSA)*, 2017.

[54] M. Åsberg, T. Nolte, M. Joki, J. Hogbrink, and S. Siwani, "Fast linux bootup using non-intrusive methods for predictable industrial embedded systems," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, Sept 2013, pp. 1–8.

[55] Texas Instruments, "Boot sequence," http://processors.wiki.ti.com/index.html.

[56] H. C. A. van Tilborg and S. Jajodia, Eds., *TCG Trusted Computing Group*. Boston, MA: Springer US, 2011, pp. 1279–1279.

[57] K. Dietrich and J. Winter, "Secure boot revisited," in *2008 The 9th International Conference for Young Computer Scientists*, Nov 2008, pp. 2360–2365.

[58] Y. Liu, J. Briones, R. Zhou, and N. Magotra, "Study of secure boot with a fpga-based iot device," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2017, pp. 1053–1056.

[59] G. Fedorkow, "What's the difference between secure boot and measured boot?" http://forums.juniper.net/t5/Security-Now/What-s-the-Difference-between-Secure-Boot-and-Measured-Boot/ba-p/281251, 2015.

[60] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015047.1015049

[61] Z. S. Huang and I. G. Harris, "Return-oriented vulnerabilities in arm executables," in *2012 IEEE Conference on Technologies for Homeland Security (HST)*, Nov 2012, pp. 1–6.

[62] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, May 2008, pp. 296–310.

[63] P. Choi and D. K. Kim, "Design of security enhanced tpm chip against invasive physical attacks," in *2012 IEEE International Symposium on Circuits and Systems*, May 2012, pp. 1787–1790.

[64] S. Glass, "Verified u-boot," https://lwn.net/Articles/571031/, 2013.

[65] The TCG community, "Trusted platform module (TPM) summary," 2008.

[66] W. Fang, C. Zhou, Y. Zhang, and L. Zhang, "Research and application of trusted computing platform based on portable tpm," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, Aug 2009, pp. 506–509.

[67] K. J. Singh and D. S. Kapoor, "Create your own internet of things: A survey of iot platforms." *IEEE Consumer Electronics Magazine*, 2017.

[68] E. Thompson, "MD5 collisions and the impact on computer forensics," *Digital Investigation*, vol. 2, pp. 36–40, 2005.

[69] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *Advances in Cryptology – CRYPTO*, J. Katz and H. Shacham, Eds. Springer International Publishing, 2017.

[70] G. Ateniese, "Efficient verifiable encryption (and fair exchange) of digital signatures," in *ACM Conference on Computer and Communications Security (CCS)*, 1999.

[71] B. Schneier, "Advanced Protocols," in *Applied Cryptography, Second Edition*. John Wiley & Sons, Inc., 2015.

[72] V. Shmatikov and J. C. Mitchell, "Finite-state analysis of two contract signing protocols," *Theoretical Computer Science*, 2002.

[73] J. Tomić and W. Kempton, "Using fleets of electric-drive vehicles for grid support," *Journal of Power Sources*, vol. 168, no. 2, 2007.

[74] S. Duquennoy, A. Elsts, B. A. Nahas, and G. Oikonomo, "TSCH and 6tisch for Contiki: Challenges, Design and Evaluation," in *IEEE Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2017.

[75] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," RFC Editor, Tech. Rep. RFC4944, 2007.

[76] A. Kurniawan, *Practical Contiki-NG Programming for Wireless Sensor Networks*. Apress, 2018.

[77] X. Vilajosana, P. Tuset, T. Watteyne, and K. Pister, "OpenMote: Open-Source Prototyping Platform for the Industrial IoT," in *Ad Hoc Networks*. Springer, 2015.

[78] S. Tozlu and M. Senel, "Battery lifetime performance of wi-fi enabled sensors," in *IEEE Consumer Communications and Networking Conference (CCNC)*, 2012.

[79] S. Dziembowski, L. Eckey, and S. Faust, "Fairswap: How to fairly exchange digital goods," in *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[80] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, 2016.

[81] Y. Hanada, L. Hsiao, and P. Levis, "Smart Contracts for Machine-to-Machine Communication: Possibilities and Limitations," *IEEE Conference on Internet of Things and Intelligence System (IOTAIS)*, 2018.

[82] C. Profentzas, M. Almgren, and O. Landsiedel, "IoTLogBlock: Recording Off-line Transactions of Low-Power IoT Devices Using a Blockchain," in *IEEE Conference on Local Computer Networks (LCN)*, 2019.

[83] S. Dziembowski, S. Faust, and K. Hostáková, "General State Channel Networks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[84] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[85] F. Gai, C. Grajales, J. Niu, M. M. Jalalzai, and C. Feng, "Cumulus: A BFT-based Sidechain Protocol for Off-chain Scaling," *arXiv*, 2019.

[86] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart Contract Templates: essential requirements and design options," *arXiv*, 2016.

[87] P. Zhang, J. White, D. C. Schmidt, and G. Lenz, "Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps," *arXiv*, 2017.

[88] B. A. Nahas, S. Duquennoy, and O. Landsiedel, "Concurrent Transmissions for Multi-Hop Bluetooth 5," in *IEEE Conference on Embedded Wireless Systems and Networks (EWSN)*, 2019.

[89] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-Based on-Line Energy Estimation for Sensor Nodes," in *Proceedings of the 4th Workshop on Embedded Networked Sensors (EmNets)*. ACM, 2007.

[90] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," *https://lightning.network/*, 2016.

[91] R. Network, "What is the raiden network?" *https://raiden .network/*, 2018.

[92] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*. Springer, 2015.

[93] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg, "Settling payments fast and private: Efficient decentralized routing for path-based transactions," *Network and Distributed System Security Symposium (NDSS)*, 2017.

[94] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "PERUN: Virtual Payment Channels over Cryptographic Currencies," *IACR Cryptology ePrint Archive*, 2017.

[95] R. Khalil and A. Gervais, "NOCUST-A Non-Custodial 2nd-Layer Financial Intermediary," *IACR Cryptology ePrint Archive*, 2018.

[96] N. Reijers and C.-S. Shih, "CapeVM: A Safe and Fast Virtual Machine for Resource-Constrained Internet-of-Things Devices," in *ACM Conference on Embedded Networked Sensor Systems (SenSyS)*, 2018.

[97] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," in *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[98] R. Müller, G. Alonso, and D. Kossmann, "A Virtual Machine for Sensor Networks," in *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[99] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding Concurrency to Smart Contracts," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 2017.