THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Self-Reliance for the Internet of Things: Blockchains and Deep Learning on Low-Power IoT Devices

CHRISTOS PROFENTZAS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

Self-Reliance for the Internet of Things:
Blockchains and Deep Learning on Low-Power IoT Devices
CHRISTOS PROFENTZAS
ISBN 978-91-7905-665-0

© 2022 CHRISTOS PROFENTZAS
All rights reserved

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5131
ISSN 0346-718X
Technical report 214D

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

This thesis has been prepared using LaTeX
Cover image has been designed using resources from Flaticon.com
Printed by Chalmers Reproservice
Gothenburg, Sweden 2022

Self-Reliance for the Internet of Things:
Blockchains and Deep Learning on Low-Power IoT Devices

CHRISTOS PROFENTZAS
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The rise of the Internet of Things (IoT) has transformed common embedded devices from isolated objects to interconnected devices, allowing multiple applications for smart cities, smart logistics, and digital health, to name but a few. These Internet-enabled embedded devices have sensors and actuators interacting in the real world. The IoT interactions produce an enormous amount of data typically stored on cloud services due to the resource limitations of IoT devices. These limitations have made IoT applications highly dependent on cloud services. However, cloud services face several challenges, especially in terms of communication, energy, scalability, and transparency regarding their information storage. In this thesis, we study how to enable the next generation of IoT systems with transaction automation and machine learning capabilities with a reduced reliance on cloud communication. To achieve this, we look into architectures and algorithms for data provenance, automation, and machine learning that are conventionally running on powerful high-end devices. We redesign and tailor these architectures and algorithms to low-power IoT, balancing the computational, energy, and memory requirements.

The thesis is divided into three parts: Part I presents an overview of the thesis and states four research questions addressed in later chapters. Part II investigates and demonstrates the feasibility of data provenance and transaction automation with blockchains and smart contracts on IoT devices. Part III investigates and demonstrates the feasibility of deep learning on low-power IoT devices. We provide experimental results for all high-level proposed architectures and methods. Our results show that algorithms of high-end cloud nodes can be tailored to IoT devices, and we quantify the main trade-offs in terms of memory, computation, and energy consumption.

Keywords: Internet of Things, TinyML, Deep Neural Networks, On-device Learning, Blockchain, Smart Contracts

List of Publications

This thesis is based on the work contained in the following papers:

(A) **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.*

(B) **Christos Profentzas**, Magnus Almgren, Olaf Landsiedel, "TinyEVM: Off-Chain Smart Contracts on Low-Power IoT Devices", *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems (ICDCS), 2020.*

(C) **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.*

(D) **Christos Profentzas**, Magnus Almgren, Olaf Landsiedel, "Performance of deep neural networks on low-power IoT devices", *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things, 2021.*

(E) **Christos Profentzas**, Magnus Almgren, Olaf Landsiedel, "MicroTL: Transfer Learning on Low-Power IoT Devices", *Proceedings of the 47th IEEE Conference on Local Computer Networks (LCN), 2022.*

(F) **Christos Profentzas**, Magnus Almgren, Olaf Landsiedel, "MiniLearn: On-Device Learning for Low-Power IoT Devices", *Proceedings of the 22' International Conference on Embedded Wireless Systems and Networks (EWSN), 2022.*

Personal Contribution

I contributed to **Paper A**, **Paper B**, **Paper D**, **Paper E**, **Paper F** as the main designer, implementer and executor of all of the experiments. Moreover, I contributed to formulating the research questions and designing the algorithms before each experimentation. The design, implementation, and execution of experiments of **Paper C** was performed in collaboration with Mirac Günes.

I led the writing of manuscripts and collaboration with other authors for all papers. The overall editorial process of **Papers A**, **Paper B**, **Paper D**, **Paper E**, **Paper F** was a contribution of me (as leading author), Magnus Almgren, and Olaf Landsiedel. The overall editorial process of **Paper C** was a contribution of me (as leading author), Mirac Günes, Yiannis Nikolopoulos, Magnus Almgren, and Olaf Landsiedel.

Acknowledgment

My interest in computer science started early in my life. I'm grateful to my teacher Giorgos, who showed me the world of computers when I was a teenager. I would like to thank my family, especially my mother Dorothea, my brother Dimitris, and my aunt Vaso for their support in carrying on my studies. I want to thank Lovisa for her love and her support during the ups and downs of my Ph.D. journey.

This work was made possible with the contribution of several people. I want to thank my supervisors, Magnus and Olaf, for their help. Next, I would like to thank the administrators, Agneta, Monica, Eva, Marianne, Rebecca, Jenny, and Clara, for their help and my manager, Tomas, for creating a good working environment. I want to thank Lars and Michael for their IT support and tools to perform my experiments. I would like to thank all of my colleagues and especially Georgia, Andreas, Dimitris, Babis, Hannah, Bastian, Huaifeng, Hamdy, Thomas, Ivan, Fazeleh, Karl, Amir, Beshr, Nasser, Aljoscha, Francisco, Kim, Shiliang, Ahmed, Marina, Romaric, Uddipana, Phillipas, Elad, Vincenzo, Petros, Stavros, Vaggelis, Yiannis N., Yiannis S., Prajith, Nadia, Waqar, Ahsen, Pedro, and Miquel that make Chalmers a fun and interesting workplace. Finally, I would like to thank my colleagues: Valentin, Patrick, Oliver, and Janek, for the excellent time that I had at Kiel University.

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 2022

Contents

Abstract i

List of Publications iii

Personal Contribution v

Acknowledgement vii

List of Figures xii

List of Tables xiv

Part I: Thesis Overview 1

1 Introduction 3

1.1 Motivation . . . 5
1.2 Application Scenario . . . 6
1.3 Background . . . 8
1.3.1 Blockchains . . . 8
1.3.2 Deep Neural Networks . . . 12
1.4 Related Work . . . 15
1.4.1 Blockchains and Smart Contracts within IoT . . . 15
1.4.2 Deep Learning on Embedded Systems . . . 17
1.4.3 Firmware Verification . . . 18
1.5 Research Questions . . . 19
1.6 Thesis Contributions . . . 20
1.6.1 Blockchain and Smart Contracts on Verified IoT Devices (Part II) . . . 21
1.6.2 Embedded Deep Learning on IoT Devices(Part III) . . . 23
1.7 Conclusion and Future Directions . . . 25

Part II: Blockchains and Smart Contracts on Verified IoT Devices **27**

2 IoTLogBlock: Recording Off-line IoT Transactions **29**
2.1 Introduction . . . 29
2.2 Overview and Background . . . 31
2.2.1 Contract Signing Protocols . . . 32
2.2.2 Smart Contracts . . . 33
2.2.3 Blockchains and Hyperledger . . . 33
2.3 Application Scenario and Adversary Model . . . 33
2.3.1 Application Scenario . . . 33
2.3.2 Adversary Model . . . 34
2.4 System Design . . . 35
2.4.1 Design Overview . . . 35
2.4.1.1 Node Discovery and Node Interaction (1*) . . 35
2.4.1.2 Interaction Between Nodes and the Cloud . . . 35
2.4.1.3 Verification and Storage in the Cloud (2*, 3*) 35
2.4.2 Setup: Deploying New Devices . . . 35
2.4.3 Creating and Signing Transactions . . . 36
2.4.4 Validation with the Smart Contract . . . 36
2.4.5 Register Transactions in the Blockchain . . . 37
2.4.6 Security Analysis . . . 37
2.5 Experimental Evaluation . . . 38
2.5.1 Implementation and Experimental Setup . . . 38
2.5.2 Evaluation of IoTLogBlock on the IoT Node . . . 39
2.5.3 System Performance of IoTLogBlock. . . . 41
2.6 Discussion and Limitations . . . 43
2.7 Related Work . . . 43
2.8 Conclusion . . . 44
2.9 Acknowledgments . . . 44

3 TinyEVM: Smart Contracts on Low-Power IoT Devices **45**
3.1 Introduction . . . 46
3.2 Background . . . 47
3.2.1 Overview of Blockchains . . . 47
3.2.2 Smart Contracts and the Ethereum Virtual Machine . . 48
3.2.3 Payment Channels . . . 48
3.2.4 Side-Chains . . . 49
3.3 TinyEVM Overview . . . 49
3.3.1 Application Scenario: Smart Parking . . . 49
3.3.2 System Requirements . . . 50
3.3.3 System Challenges . . . 50
3.3.4 Threat Model . . . 51
3.4 TinyEVM System Design . . . 51
3.4.1 On- and Off-Chain Transactions: Three Phases . . . 52
3.4.2 Customized Ethereum Virtual Machine . . . 53
3.4.3 On-Chain Smart Contract . . . 54
3.4.4 Off-Chain Smart Contract . . . 54
3.4.5 On-Chain Commit & Challenge Period . . . 56

3.5 Security Analysis . . . 57
3.6 Performance Evaluation . . . 57
3.6.1 Experimental Setup . . . 58
3.6.2 Ethereum Virtual Machine on IoT Devices . . . 58
3.6.2.1 Memory Requirements . . . 58
3.6.2.2 Deployment Execution Time . . . 60
3.6.3 Off-chain Payment Channels . . . 61
3.6.3.1 Memory Requirements. . . . 61
3.6.3.2 Cryptographic Modules . . . 62
3.6.3.3 Energy Consumption . . . 62
3.7 Related Work . . . 63
3.8 Conclusion . . . 65
3.9 Acknowledgments . . . 66

4 Performance of Secure Boot in Embedded Systems 67
4.1 Introduction . . . 67
4.2 Background and Definitions . . . 69
4.3 Adversary Model . . . 70
4.4 Systems Overview . . . 72
4.4.1 Software-Based Secure Boot with U-Boot . . . 72
4.4.2 Hardware Security for Secure Boot . . . 74
4.5 Evaluation . . . 75
4.5.1 Experimental Setup . . . 75
4.5.2 Evaluation Results . . . 75
4.5.2.1 Software Mechanism of U-Boot . . . 76
4.5.2.2 TPM Hardware on BeagleBone . . . 77
4.5.3 Discussion . . . 78
4.6 Limitations and Discussion . . . 78
4.7 Conclusion . . . 79
4.8 Acknowledgments . . . 79

Part III: Embedded Deep Learning on IoT Devices 81

5 Performance of DNN on Low-Power IoT Devices 83
5.1 Introduction . . . 83
5.2 Background and Motivation . . . 84
5.2.1 Convolutional Neural Networks . . . 84
5.2.2 Quantization . . . 85
5.2.3 IoT Deep Learning Frameworks . . . 85
5.3 Benchmark Design . . . 86
5.3.1 Overview and Evaluation Goals . . . 86
5.3.2 Benchmark Process . . . 86
5.4 Performance Evaluation . . . 86
5.4.1 Experimental Setup . . . 88
5.4.2 Data Sets . . . 88
5.4.3 Convolutional Neural Networks . . . 88
5.4.4 Benchmark Evaluation . . . 89
5.4.5 Discussion and Limitations . . . 91
5.5 Related Work . . . 92

5.6 Conclusion 92

6 MicroTL: Transfer Learning on Low-Power IoT Devices 95
6.1 Introduction 95
6.2 Background 97
6.2.1 Transfer Learning 97
6.2.2 Quantization 97
6.3 MicroTL Design 98
6.3.1 System Challenges 98
6.3.1.1 Learning After Quantization 98
6.3.1.2 Device Resource Constraints 99
6.3.2 MicroTL Overview. 99
6.3.3 Dequantization. 100
6.3.4 Pre-trained Neural Networks 100
6.3.5 Sample Storage 100
6.3.6 MicroTL Training Algorithm 101
6.4 Performance Evaluation 102
6.4.1 Experimental Setup 102
6.4.2 Training Sets and Neural Networks 102
6.4.3 MicroTL Learning Accuracy 104
6.4.4 MicroTL Performance on Low-Power IoT Devices 106
6.4.5 Discussion and Limitations 107
6.5 Related Work 108
6.6 Conclusion 109

7 MiniLearn: On-Device Learning for Low-Power IoT Devices 111
7.1 Introduction 111
7.2 Background & Related Work 113
7.3 MiniLearn Overview 115
7.3.1 Application Scenario 115
7.3.2 Data Flow 115
7.3.3 Training Steps 115
7.3.4 System Challenges 115
7.3.5 MiniLearn Design 117
7.3.6 MiniLearn Architecture 117
7.3.7 MiniLearn Stages 118
7.3.7.1 Dequantizing and Pruning of Filters (Stage-I) 118
7.3.7.2 Training the Filters (Stage-II) 118
7.3.7.3 Fine-Tuning (Stage-III) 119
7.4 Experimental Methodology 119
7.5 Results & Discussion 120
7.5.1 MiniLearn Accuracy Results 121
7.5.2 MiniLearn Performance on IoT Devices 122
7.5.3 Discussion and Limitations 124
7.6 Related Work 125
7.7 Conclusion 126

Bibliography 131

List of Figures

1.1 Thesis Overview 4
1.2 IoT Ecosystem 5
1.3 Blockchain Application Scenario 7
1.4 Network Topologies 8
1.5 Blockchain Main Structure 9
1.6 Example of Merkle Tree 9
1.7 Smart Contract Execution 11
1.8 CNN Architecture 12
1.9 RNN Architecture 12
1.10 Magnitude of Research Questions 20

2.1 Car Rental Scenario 30
2.2 Signature Exchange Protocol 32
2.3 Hyperledger-Fabric Architecture 34
2.4 Electric Current Draw from Contract Signing 40
2.5 Off-line Transaction Evaluation 42

3.1 TinyEVM Application Scenario 50
3.2 TinyEVM System Design 52
3.3 Memory Usage of Smart Contracts 59
3.4 Deploying Time of Smart Contracts 60
3.5 Electric Current Drawn During Off-chain Payments 63

4.1 Boot Process of Embedded Systems 69
4.2 U-Boot Configuration 72
4.3 Boot Sequence of U-Boot 73
4.4 Secure Boot with TPM 74
4.5 The Evaluation of U-Boot 76
4.6 Evaluation of Entire Boot Process 77

5.1 Convolution Neural Network Example 85
5.2 Benchmark's Development Process 87

5.3 RAM Footprints of Benchmarks 89
5.4 Flash Footprints of Benchmarks 89
5.5 Electric Current Drawn During Inference 90

6.1 MicroTL Approach 99
6.2 Dequantization 100
6.3 Training Evaluation of MicroTL 104
6.4 Performance Evaluation of MicroTL 105

7.1 A representative architecture of a Convolution Neural Network (CNN) typically found on low-power IoT devices. The CNN consists of several layers stacking together: convolution kernels, max pooling, and fully connected (flatten & dense). The final layer is the output for the classification task. 113
7.2 Structure of Minilearn 116
7.3 Average accuracy using MiniLearn with different pruning percentages on KWS 7.3a, CIFAR 7.3b, and WISDM 7.3c. The original baseline is the pre-trained network test on the sub-set, and *cloud* represents sending data and retraining in the cloud. The shaded area represents the standard deviation. 121
7.4 MiniLearn performance during training in terms of time and memory consumption (using 600 samples). We apply different pruning percentages with a 25% step. 122
7.5 Electric current drawn during training on nRF-52840-DK, by applying at 3 V, during a complete training with 100 samples on CIFAR-subset. 122
7.6 Inference time and memory consumption of the neural network after applying MiniLearn with corresponding accuracy (trained on 600 samples). We apply different pruning percentages with a 25% step. 123

List of Tables

1.1 Research Questions 20

2.1 Memory Footprint of IoTLogBlock 40
2.2 IoTLogBlock Cryptographic Operations 41
2.3 IoTLogBlock Energy Consumption 41

3.1 TinyEVM Specifications 53
3.2 Overview of Deployed Smart Contracts 61
3.3 Memory Footprint of TinyEVM 61
3.4 Energy Consumption of TinyEVM 62
3.5 Performance of Cryptographic Operations on TinyEVM 64

4.1 Secure Boot Terminology 71
4.2 Secure Boot Hardware Specifications 75
4.3 Verification Overhead of TPM 78

5.1 High-level Comparison of Deep Learning Frameworks 85
5.2 Benchmark on MNIST 87
5.3 Benchmark Evaluation using MNIST 91
5.4 Benchmark Evaluation using CIFAR-10 91

6.1 Pre-Trained Neural Networks 103
6.2 Accuracy of MicroTL 103
6.3 Complete Evaluation of MicroTL 108

7.1 Networks used in the experiments. The table shows the shape size, computations in terms of multiply-accumulate (MAC), and the layer output in Bytes for each layer. 129
7.2 MiniLearn training evaluation, without pruning, on CIFAR and KWS subsets. We report the RAM and Flash memory footprints, the average training time, and average energy consumption during the complete training (including fine-tuning). 130

Part I

Thesis Overview

CHAPTER 1

Introduction

In recent years, we have experienced the rapid adoption of the Internet of Things (IoT), where millions of Internet-enabled sensors and actuators communicate and interact with each other. These interactions produce an enormous amount of data typically stored on cloud services due to the resource limitations of IoT devices. Their communication capabilities and the accumulation of IoT data in the cloud have led to two major trends. The first trend is the next generation of automated IoT systems and applications (e.g., autonomous cars, industrial robots, smart homes, and appliances), where Internet-enabled embedded devices interact and make decisions using cloud services with physical objects. Second, the trend of data analytics has emerged, where cloud services discover statistical relationships on very large, diverse datasets driven by advances in machine learning (ML) and IoT systems.

However, these trends have made IoT applications extensively reliant on cloud services. Centralized cloud-based architectures raise several questions on the scalability, transparency, and availability to process the enormous amount of IoT data. Moreover, our societies set strict requirements regarding the privacy of the data and trustworthiness (i.e., non-repudiation) in order to integrate automated systems in daily-life activities. Arguably the success of the next generation of IoT applications is related to the ability of wireless low-power devices to learn from the environment, make predictions, and automate agreements with reduced cloud interaction. Thus, the next step for IoT systems is self-reliance and decision-making utilizing parts of data locally. This thesis takes on the challenge of empowering IoT devices with capabilities typically found on powerful cloud nodes for transaction automation, data provenance, and machine learning.

The primary motivation comes from two upcoming technologies able to become the subsequent enablers for the next generation of IoT applications by overcoming the limitations of cloud services: (I) Blockchains that support decentralized storage in a transparent way and automated program execution (i.e., smart contracts) when predetermined conditions are met. (II) TinyML,

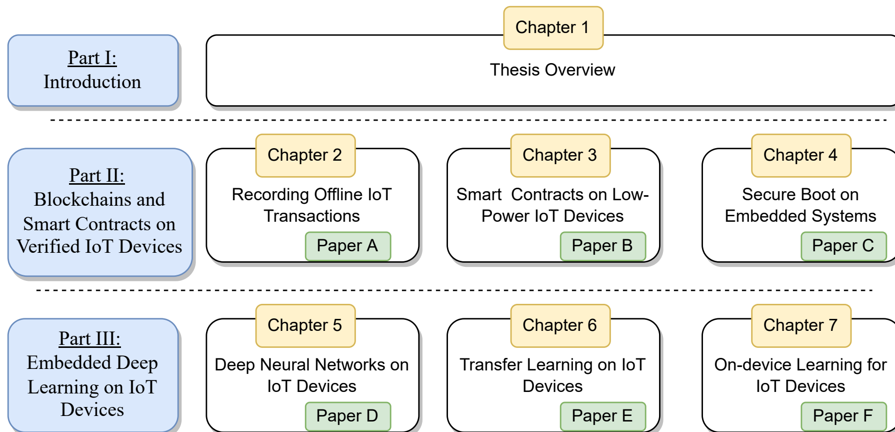
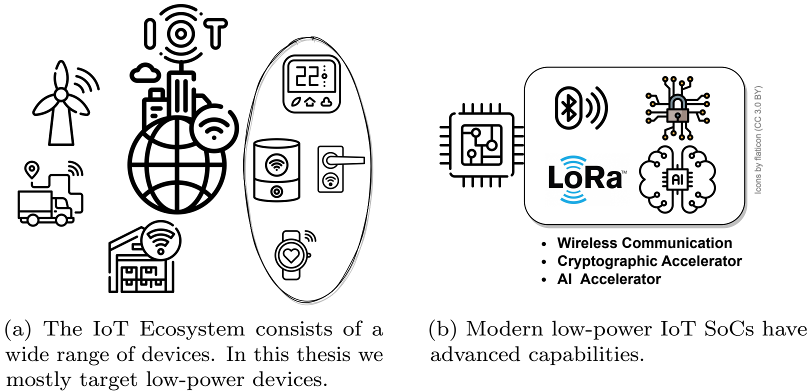


Figure 1.1: An overview of the thesis. It includes seven chapters divided into three parts, the first part is the introduction, the second part demonstrates the integration of blockchains and smart contracts on verified IoT devices, and the third part presents the tailoring of deep learning to IoT devices.

where Machine Learning (ML) algorithms run on low-power embedded devices. However, these emerging technologies set an increased pressure on existing challenges of resource-constrained devices in terms of energy, memory, and computation. For example, IoT devices should operate for long periods with wireless communication based on limited power resources, typically batteries or energy harvesting technologies. Using verification processes (i.e., cryptographic capabilities) and ML capabilities commonly demand energy-hungry operations that will impact the strict power budget of a typical IoT device. Moreover, the limited memory of IoT devices restricts the storage for potential transactions or learning on the device. Finally, IoT devices lack the high-end processing capabilities required to run intensive data processing and cryptographic operations, typically used by blockchain and machine learning algorithms.

This thesis presents research studies regarding the feasibility of incorporating local data in machine learning and automated agreements on low-power IoT devices by tailoring algorithms typically found on powerful high-end devices. To achieve this, we propose, implement, and evaluate system designs on resource-constrained IoT devices while balancing computational, energy, and memory consumption. The thesis has three parts: Part I presents an overview of the thesis and states four research questions addressed in later chapters. Part II considers how data provenance (with blockchains), trust, and transaction automation algorithms (i.e., smart contracts) can be adapted to IoT devices. Part III considers how deep learning algorithms can be tailored to IoT devices. The overview of the thesis, along with the chapters and papers, is illustrated in Figure 1.1.

Introduction outline. The rest of the introduction is organized as follows. Section 1.1 describes and explains in detail the new opportunities and challenges introduced by enabling novel algorithms on low-power IoT devices. Section 1.2 provides a motivational scenario on how the next generation of IoT devices interact with each other in the real world. Section 1.3 provides



(a) The IoT Ecosystem consists of a wide range of devices. In this thesis we mostly target low-power devices.

(b) Modern low-power IoT SoCs have advanced capabilities.

Figure 1.2: The landscape of IoT ecosystem and modern capabilities of System on a Chip (SoC) for low-power IoT devices.

the essential background for the main enablers presented in the motivational scenario. Section 1.4 provides the related work and the aspects that state-of-art does not address, which provides the main motivation for this thesis. Section 1.5 presents the research questions and challenges in providing IoT systems capabilities typically found on powerful high-end devices. Section 1.6 states the contributions of the thesis and how each chapter responds to each research question. Finally, Section 1.7 provides the conclusion remarks and future directions of the thesis.

1.1 Motivation

The field of the Internet of Things (IoT) has tremendously evolved over the years, from simple wireless sensors and actuators to complex systems with several dedicated sub-systems. The integration of Internet-enabled devices has opened the door to new applications, in areas such as intelligent transportation [1], smart homes [2], wearables [3], connected healthcare [3], and smart cities [4], to name but a few. As we can see in Figure 1.2a, the IoT ecosystem consists of different types of IoT devices. This thesis mostly focuses on IoT systems that utilize low-cost embedded devices with low-power consumption that are able to operate autonomously for years. These devices experience significant constraints in terms of memory, energy, and computational capabilities. Moreover, embedded devices often have tight latency requirements with real-time properties. However, to enable the next generation of smart applications with low-cost, low-power IoT devices, we need to integrate new paradigms of data provenance, transaction automation, and machine learning algorithms.

Data provenance [5] [6] and transaction automation [7] is particularly important when considering how frequently IoT devices interact and produce records of interactions in open environments. IoT applications need to store multiple interactions and agreements among multiple IoT devices, but the

devices need to establish trust with each other before signing any agreement. It is challenging to establish trust in a transparent way as different entities own or operate the IoT devices. In practice, IoT applications utilize cloud services as a central authority to store all data and records of transactions. However, there is a lack of transparency on how cloud providers can use, store, and share information among entities with a conflict of interest. Moreover, these centralized cloud architectures need to establish resource-intensive secure communication [8] between the devices and suffer from a single point of failure [5]. Although parts of the verification process can be integrated on local resource-constrained devices, as cryptographic accelerators have become readily available on low-cost, low-power IoT devices (illustrated in Figure 1.2), the device verification capabilities have not been widely explored.

Another direction that we explore in this thesis is the recent advances in machine learning. Even though machine learning is mainly associated with cloud services, we see aspects of machine learning gradually integrated into IoT systems [9–11]. This development can enable the next generation of smart applications by moving capabilities of data analytics and machine learning algorithms closer to IoT devices. Machine learning on IoT can enable anomaly detection [12], human activity recognition [13], computer vision [14], and keyword spotting [15] by incorporating sensor data such as images, text, and audio. Specifically, machine learning inference can augment or replace manual processing of IoT data and speed-up or improve decision-making. In Figure 1.2, we can see that modern IoT System-on-Chips (SoCs) (e.g., ARM and RISC-V architectures) considerably foster deep neural network accelerators (e.g., NPUs [16], TPUs [17]) along with security and cryptographic capabilities (e.g., TrustZone [18], CryptoCell [19]) integrated on the same SoC. However, current IoT systems still depend on centralized cloud-based architectures to provide machine learning models or offload data. The only way for an IoT device to acquire or modify a machine learning model is to wait for cloud services to train a new model, optimized for the local hardware [9, 10, 20, 21], and re-download it. However, IoT devices are deployed and interact in real-world environments, and the initial problem considered when training a model can change [22]. The current architectures do not consider cases where users on the fly might want to add or remove a class from an existing model or optimize the model for a sub-class problem.

1.2 Application Scenario

We provide an IoT parking application, illustrated in Figure 1.3 to show the interactions of IoT devices in a real-world scenario. Finding and analyzing the cost of a parking place is not a trivial task in a busy modern city. The driver needs to cognitively think about the location, price, and parking availability to determine the value of a parking slot. Similarly, parking companies have to analyze a significant amount of data from parking sensors to charge their fee effectively. In this example, IoT devices interact to negotiate parking and electricity fees. With modern IoT ecosystems, smart parking lots are equipped with sensors that can detect occupation and interact with a vehicle occupying a spot, for example, via low-power wireless technologies. As the vehicle may

Icons by flaticon (CC 3.0 BY)

Figure 1.3: An overview of an application scenario based on the integration of blockchain and embedded machine learning. The user may interact with a smart wallet to give a command. Next, the smart wallet negotiates the fees with an electric charger. The electric charger needs specific information about the car (e.g., battery safety standard). The scenario involves entities that own or operate different parts of the devices.

belong to a rental company, the electric charger and the driver may have a conflict of interests, and the settlement of the fees becomes a complicated process.

It is vital for IoT devices to interact with a local context (e.g., sensor data) and not depend on cloud services to settle the fee every time the IoT devices negotiate a parking fee. Moreover, the scenario includes a smart wallet by using keyword spotting to recognize commands from the user. The smart wallet has a pre-trained model based on keyword spotting of generic actions. However, after using the device for a while, the local user includes different commands specialized to her voice and accent. While the user engages in activities, the device creates a training set from local data and labeling either by using another sensor or by asking the user through device interaction. The user can optimize and automate the fee negotiation using the personalized smart wallet. Overall, we envision a marketplace where rental vehicles, parking companies, and the drivers negotiate the terms of parking and perform a form of micropayments.

In practice, as we can see in Figure 1.3, when a vehicle approaches the parking lot, the devices will come within communication range via low-power wireless technologies. The parking lot can initiate an automatic program (i.e., a smart contract) and negotiate and register a payment. The devices may do multiple transactions and interactions during the parking, such as hourly payments or updates on the payment rates based on the time of day. At the end of the parking period, we would like the devices to agree and sign the final state without constantly initiating the blockchain services to settle the agreement. Finally, the devices are exposed to security threats through the Internet and short-range communication (e.g., Bluetooth). For example, a malicious user can find ways to connect to a device and modify its software, for

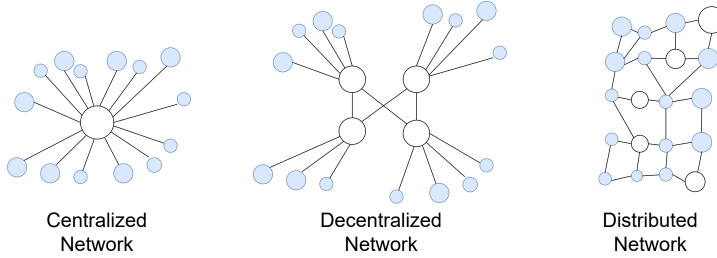


Figure 1.4: An illustration of the different topologies for connected nodes: centralized, decentralized, and distributed.

example, the charging ECU of the electric car or the smart wallet. Ensuring the integrity of the software is an essential property to enhance trust in IoT devices.

This thesis primarily targets the resource-constrained devices involved in the above scenario, like the smart wallet and parking sensor, and it examines how to enable the above scenario without relying on a constant communication channel with cloud services. The feasibility of the above application scenario is motivated by three facilitators:

- Blockchain integration that enables distributed data storage and self-reliance of participants.
- Smart contracts that utilize sensor reading and actuators as part of IoT transactions.
- Inference and machine learning that is executed locally on IoT devices personalized to the end-user.

1.3 Background

The previous section outlines the main characteristics of the enablers we consider in this thesis for self-reliance. This section provides an overview of the enablers in two parts. First, we provide an overview of methods for transaction automation and recording IoT information in a transparent way. Particularly, we discuss blockchain and smart contracts and the challenges of integration with low-power IoT devices. Second, we provide an overview of methods for deep learning and inference on IoT devices. Deploying deep neural networks on IoT devices has particular challenges, such as quantization and optimization. We provide a basic overview of the most common techniques.

1.3.1 Blockchains

In recent years, broad studies on blockchain technologies reveal new architectures and application scenarios [6, 23]. The intuition behind most of the studies is the replacement of the central authority typically found in cloud-based paradigms. The data flow of cloud-based paradigms [24], as depicted in

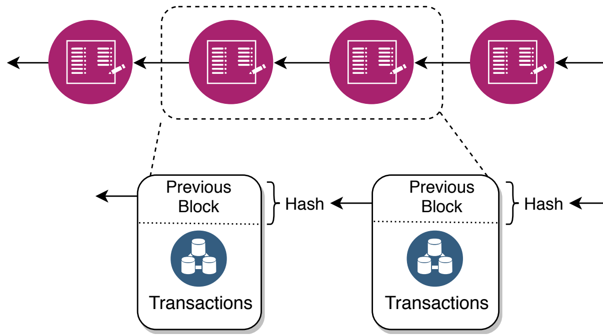


Figure 1.5: The main structure of blockchain is an appended link-list by using the cryptographic hash value of the previous block.

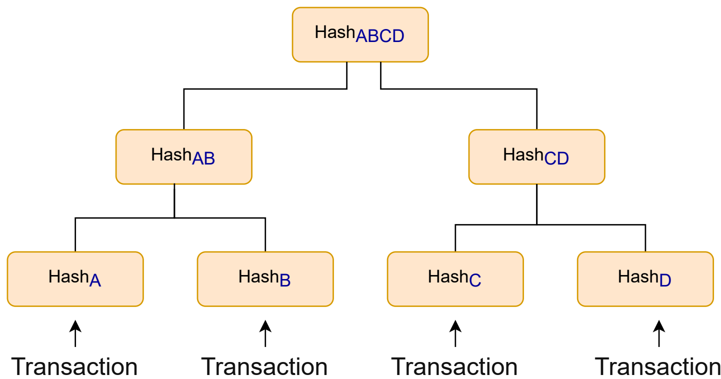


Figure 1.6: A simplified illustration of the Merkle tree. The top hash value is referred to as the root hash of the tree. Each hash value of a transaction is concatenated with others in a binary tree structure. In practice, a tree may have hundreds of transactions.

Figure 1.4, is concentrated to one (centralized) or a limited number of nodes (decentralized). On the other hand, the blockchain acts as a distributed ledger [25] maintained by a collection of distributed nodes (as shown in Figure 1.4) communicating together and replicating the same data structure. Blockchain has no central authority to govern the data flow, there are some special nodes to verify the new data blocks. The data structure of blockchain, visualized in Figure 1.5, is an append-only linked list that chains together blocks of transactions. Each block is connected with the previous block by using the cryptographic hash value. Any change to a previously chained block inherently changes the hash value of all other blocks. As the blockchain is replicated to all nodes in the system, it provides fault tolerance, and each node can verify and validate all chained blocks without contacting any central authority.

Merkle Trees. Distributed ledgers such as blockchains are becoming significantly large as new blocks are appended over time. Without careful

design, verifying and encoding information in the blockchain structure becomes a rigorous task. In cryptocurrencies [26,27], for example, there are millions of transactions that need to be verified. For that reason, blockchains commonly utilize Merkle trees [28] to organize their data. An example of a Merkle tree is illustrated in Figure 1.6. With a Merkle tree, instead of obtaining a hash value for each transaction, pairs of transactions (tree leaves) are concatenated and hashed together in a binary tree structure until there is one hash value for the entire block at the root of the tree. The Merkle tree is based on the cryptographic property that it is computationally infeasible to find other values with the same hash value as the root. The benefit of the Merkle tree becomes apparent when considering that a user would like to verify a specific transaction without downloading the whole blockchain (Bitcoin is over 395.5 GB during May 2022). The user only needs to verify that the root of the tree of her transaction is part of the blockchain.

Consensus One important issue of blockchain is to reach a consensus on a single state of the network. There are different proposals for reaching a consensus among the participant nodes. For example, Bitcoin [26] uses the Proof of Work (PoW) algorithm. In the PoW algorithm, the consensus is based on the participants' feasible amount of work towards solving a specific mathematical puzzle. The mathematical puzzle is based on the one-way cryptographic hash property that is computationally infeasible, given a specific output to invert the hash value and find the input. In PoW, the participants need to find a specific output with a certain amount of leading zeros, given all the previous blocks. The only way to solve it is to try a significant number of different input combinations. The effort needed to solve the puzzle prevents users from acting maliciously and incentivizes the participants by rewarding them when they actually solve the puzzle. When a node solves the mathematical problem, it creates and broadcasts a new block to the network for validation. Many other alternative algorithms have been proposed to reach a consensus in a blockchain, such as Proof of Stake (PoS) [29], Proof of Elapsed Time, and Practical Byzantine Fault Tolerance (PBFT) [30]. Depending on who can be a special node that adds new blocks to the system, we can categorize blockchains into public and private ones.

Public and private blockchains. Public blockchains [26,27] allow any node to join and contribute to the consensus process by proposing or validating new blocks to the blockchain. For example, in Bitcoin [26], participating in Proof of Work is open to everyone and provides a reward (Bitcoins) to nodes that solve the PoW puzzle. Private blockchains [31] allow only selected and verified nodes in the consensus process. Private blockchains control who joins the network and executes the consensus protocol, and there is no need for Proof of Work. The consensus can be reached using the standard PBFT [30].

Smart contracts. The realization of blockchain was initially only for storing plain transactions [26], but later it was extended with executable scripting programs (smart contracts [23]) replicated as part of the distributed ledger. Smart contracts [27] can represent financial agreements in digital form for untrusted parties. By initiating a transaction in the blockchain, and if certain parameters are fulfilled, the smart contract automatically executes all or parts of the agreement defined by the scripting code. In order to achieve this, the untrusted parties generate a public key [32] stored together with the

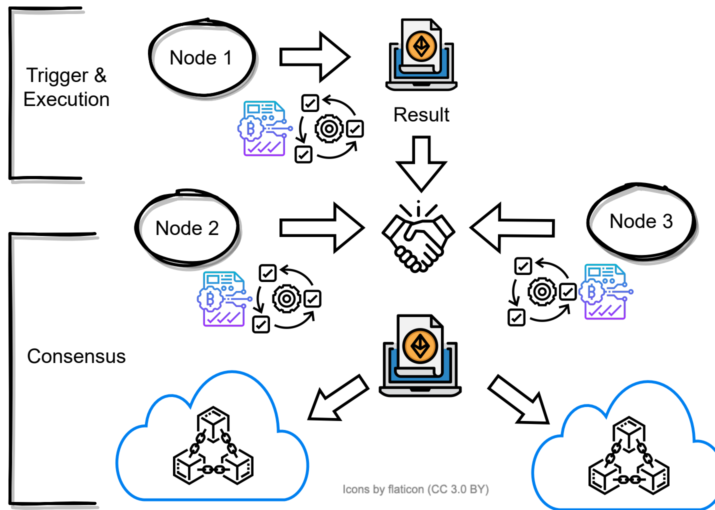


Figure 1.7: Example of smart contract execution. After the trigger of a smart contract, each node in the blockchain will execute and verify the correctness of the results. After the result is verified, it becomes part of the blockchain.

smart contract in the blockchain. Smart contracts can have variables [33] and store different states inside the blockchain, and their execution may update a previous state of a blockchain. Smart contracts are usually written in a high-level programming language [27], and only the executable bytecode is stored in the blockchain. The nodes participating in the consensus process execute and validate each triggered smart contract. The nodes use a common Virtual Machine (VM) to execute the bytecode and validate the same execution output across the network.

Even though there is significant progress on VMs for smart contract execution, they are mostly based on sequential execution with limited support for parallel or concurrent execution. Moreover, smart contracts have restricted data access outside the blockchain network. In this context, there are two obstacles when tailoring smart contracts to IoT devices. First, current virtual machines for smart contracts do not provide any opcode to handle data using sensors and actuators of IoT devices. Second, current smart contract architectures are based on virtual machines, which are commonly quite memory-intensive for resource-constrained devices such as IoT devices.

Data provenance. With data provenance, we relate the property of providing historical records of the data and their origins that connects the input and entities that can impact, create and modify the data. Several architectures [5] [6] have been proposed to achieve data provenance using blockchain technology. The tampered-proof property and the distributed nature of the blockchain allow the fast verification of the origins of the data that can be used in various scenarios of IoT systems.

Open challenges. There are three main open challenges when considering integrating IoT systems into blockchain architectures. First, there are trade-offs between the basic operations of blockchain and power, memory, and energy

consumption. The consensus algorithms require extensive processing power from participating nodes, and the ever-growing blockchain requires sufficient memory to store the complete data structure. Second, blockchains have inherent issues regarding the verification times and throughput of the transactions. Even though there are several proposals to scale the blockchains, such as sharding [34, 35], side-chains [33], payment channels [36, 37], and payment networks [38, 39], they are developed in separation without considering a common system. IoT systems consist of billions of devices, and the fragmentation of the payment systems creates interoperability issues. Third, blockchains have limited access to data outside of their network. Blockchains are only assuring the immutability of the data, but they cannot provide ways of interacting with physical objects. In order to make blockchains and smart contracts capable of providing services in the real world, external entities need to verify the real-world facts and provide that information to the blockchain. However, IoT devices depend on a local context (e.g., sensor data) to recognize that certain conditions have happened in the physical world (e.g., parking occupation). There is a significant gap between high-level blockchain architectures and the need for a local context of IoT devices.

1.3.2 Deep Neural Networks

Deep neural networks [40] are a specific family of machine learning algorithms based on computations performed by many functions or layers. The input to the neural network is considered as the first layer, with several hidden layers performing non-linear transformations and usually a final linear output providing the task for the network, e.g., classification predictions. Two prominent architectures are commonly used depending on the hidden layers performing the transformations: Convolutional Neural Networks (CNNs) [41] and Recurrent

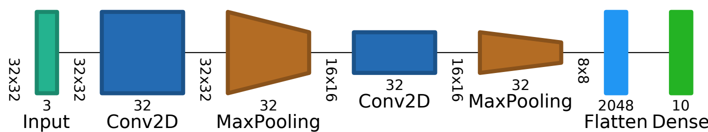


Figure 1.8: A representative architecture of a Convolution Neural Network (CNN). The CNN consists of several layers stacking together: convolution kernels, max pooling, and fully connected. The final layer is the output for the classification task.

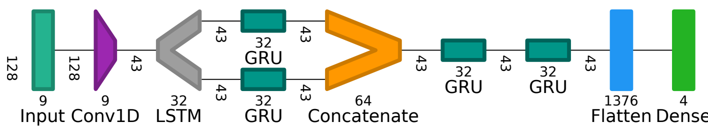


Figure 1.9: A representative architecture of a Recurrent Neural Network (RNN). The RNN consists of several layers stacking together: Convolution1D for parsing input data. LSTM(s) and GRU(s) act as memory units. Finally the fully connected layers provide the output for the classification task.

Neural Networks (RNNs) [42]. The architectures are illustrated in Figure 1.8 and 1.9.

Training and back-propagation The most widely used algorithm for training deep neural networks is back-propagation (backprop [40]). With backprop, we are learning the weights of a multi-layer neural network by employing a variant of gradients descent in the attempt to minimize the object function commonly referred to as the loss function or error of the neural network. The error is computed between the output of the neural network and the target value or label of the input. Backprop computes the gradient of the loss function with respect to the weights for each layer of the network. In order to achieve this, backprop uses the chain rule, computing the gradient one layer at a time, starting from the output layer backwards to the first layer. The updates of the weights with the corresponding gradient introduce different optimization options. The simplest version is to use stochastic gradient descent and utilize a single input-output pair by computing the gradients of the loss function with only one example. In practice, most training algorithms compute the gradient for a batch input (e.g., 32-256 samples) and update the weights after each batch instead of a single input. Common optimizations include Momentum [43] where each update of weights takes into account the gradient accumulation of previous steps and adaptive learning methods, such as AdaGrad [44] and RMSProp [45].

Generalization and overfitting. Two important topics in supervised machine learning that are closely related to each other are generalization and overfitting [40]. When we train a model, we give a set of training labeled data in order to minimize the loss function or error of the network as described earlier. However, in machine learning, our ultimate goal is not the reduction of training error but rather the model's generalization error to unseen data. When the predictions of a machine learning model achieve high results on training data but perform poorly on new unseen data, we say that the model overfits to the training data. Common methods to achieve low generation error and prevent overfitting in deep neural networks is early stopping [40], weight decay [46], and regularization [47].

Quantization. Due to IoT devices' memory, computational, and energy constraints, it is common to compress deep neural networks before deployment. The most common method is fixed-point quantization [48], where the floating-point values of the weights of the network are mapped to fixed-length integer values. For example, an int-8 quantization [49] method will utilize 8-bit fixed-point representation of original floating-point values. Quantization reduces the size of the network without changing the original architecture of the network. The method needs to re-scale the range between the minimum and maximum value of the weights and shift the zero-point as it may be different from real numbers after quantization. The pre-quantized network still needs to be trained in high-level libraries on high-end computers. This has led to two main variations of quantization: post-training and quantization-aware training.

Post-training quantization. In this widely used method [9, 10, 50], the deep neural network is trained as usual in floating-point using a standard available library (e.g., Tensorflow, PyTorch). After the training period, we can use static quantization on the network by finding a new min-max range for integer weights based on already-trained layers. This method may need

a representative data set to calibrate and fine-tune the weights based on the outputs of the activation layers. This fine-tuning is a one-time process before the deployment on the IoT device.

Quantization-ware training. This method [21, 48] simulates the quantization effect during training by duplicating all the weights and bias, one in 8-bit quantized for the forward pass, and one in 32-bit floating-point for the backward pass, together with meta-data regarding rounding effects. During the forward-pass, it utilizes the 8-bit weights of the network, but during the back-propagation, it calculates the gradient using the float-point weights. At the end of each iteration, it quantizes the floating-point weights to 8-bit values. The quantization error is part of the learning process, and it can provide better accuracy than the post-train quantization described above. However, the quantization parameters become part of the overall training hyperparameters that the developer needs to tune accordingly. Finally, the edge or cloud service provides only the completed-quantized versions of the network. Usually, essential training meta-data are not available regarding the model optimization (e.g., rounding and gradient's dynamic range) used for the targeted hardware, as the design purpose is to apply only inference on the IoT device.

Transfer Learning. Transfer learning [51] aims to reduce the amount of training data and speed up the training process. The idea is to utilize existing neural networks pre-trained on large data sets for a generic problem domain referred to as the **source domain** and adapt for similar smaller domains referred to as the **target domain**. Transfer learning uses the hidden layers of a pre-trained network as feature extractors from the source domain to append and train end-layer(s) on the target domain. A typical approach to transfer learning consists of three steps. First, it removes the Fully Connected (FC) layers at the end of a pre-trained network (e.g., Flatten & Dense in Figures 1.8 & 1.9). Second, it appends a new FC layer(s) with output matching the number of the target domain classes. Third, it freezes the weights (no backward calculations) of all layers prior to FC layer(s) and trains the network using the target domain data set by updating only the weights of FC layer(s).

Open challenges. There are three main open challenges of deep learning for IoT applications. First, it is essential to quantize the neural networks to fit on constrained embedded hardware, but current approaches assume that training and optimization occur in the cloud. These approaches lead to a significant optimization opportunity missed: *on-device learning specialized for locally collected real-time IoT data.* Second, IoT applications need to gather training data from resource-constrained devices that depend on energy-efficient communication with limited reliability and network bandwidth. In practice, this may delay or even prohibit the transmission of essential training samples from the device. Third, as cloud services struggle to address privacy concerns, they may aggregate data from different sources, where keeping independent and identically distributed (iid) samples and efficient communication is an open challenge [52].

1.4 Related Work

This section presents and summarizes the related work relevant to the blockchain, deep learning algorithms, and firmware verification of IoT systems. We contrast related work with aspects that the state-of-art does not address, which relates to the thesis's motivation and contributions.

1.4.1 Blockchains and Smart Contracts within IoT

The benefits of integrating IoT devices with blockchain have been stated in several studies [53–55]. For example, architectures such as AGasP [56] and Edgechain [57] offer IoT applications to perform tasks or exchange assets on behalf of users automatically by utilizing blockchain and smart contracts. The append-only property of blockchain ensures the transparency of transactions recorded by IoT devices. Moreover, the distributed nature of blockchains allows fault tolerance and network failures as long as a network of nodes exists. Furthermore, smart contracts [58] allow applications to enrich automated decisions and agreements in terms of financial transactions.

Another way that blockchain utilization has been proposed in IoT systems is access management. For example, Novo [59] introduces access management on IoT devices (e.g., device A wants to have access to the velocity sensor of device B) by facilitating a blockchain platform. Another architecture is IoST [60] to monitor and self-configure IoT devices. The architecture deploys a blockchain on edge computing for managing the configuration of the devices. Shafagh et al. [55] propose to decouple the data and control plane for IoT devices. The architecture targets the access control of local data and decouples the centralized cloud dependency by utilizing blockchain-based storage for sharing data among IoT devices. The design is mostly focused on data streams and data sharing on time-series IoT data at the edge. In this thesis, we focus on constrained devices in the range of 32-64MHz and 32-256 KB of RAM, which is a difference of one order of magnitudes in terms of performance. 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.

Fair exchange. One of the major issues of financial agreement using internet services is how to achieve a fair exchange of digital value. A fair exchange protocol allows the exchange of digital goods for a fixed price between a seller and buyer only if the buyer actually pays the fair agreed price. The protocol is said to be secure if the buyer only pays when she actually receives the digital goods. FairSwap [61] proposes the use of smart contracts for fair exchange, avoiding costly solutions like zero-knowledge proofs [39]. The smart contract acts as an external authority that can finalize the exchange and resolve any disagreement.

OptiSwap [62] looks into the optimization of smart contracts that act as a trusted arbiter and fairly resolves disputes if a seller and a buyer disagree. The protocol can include a default optimistic mode, assuming that the parties are aware of each other and finish the digital exchange with minimal interaction. In case of dispute, they need to interact with the arbitration part of the smart contract. Moreover, OptiSwap proposes a protection mechanism against attacks

where an adversary consistently tries to violate the protocol's agreement and fairness property based on fees paid by both but only compensates the honest party.

Payment channels & networks. A key issue of blockchain arranging financial agreement is its time to reach a consensus. In most cases of IoT applications, it is desirable to achieve instant payments with low waiting times. In terms of payment scalability, blockchain face challenges in increasing the number of transactions per second and the time to complete each transaction. In this way, Payments Channels [63] have been designed as an instant payment protocol without the need for arbitration. Payment channels decouple the need to register each transaction in the blockchain by allowing off-line transactions and registering only two finalized transactions in the main blockchain. Furthermore, researchers have proposed several 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 [64] was one of the first implementations of such networks for Bitcoin. The equivalent network for Ethereum is Raiden [65].

There are three main challenges to making payment channels usable in the context of IoT systems. One challenge is to open a payment channel in both directions. Duplex micropayments [66] are an extension to allow the user to have this type of payment channel. Second, a user cannot reallocate the locked money in the payment channel. Revive [36] allows a user to rebalance the payment channels and reallocate money to a channel without the cost of closing and reopening it. Third, there are privacy concerns regarding the ability to track payments. Bold [37] 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 [67], and SpeedyMurmurs [68] are using distributed architectures. However, the above approaches assume active communication and synchronization among nodes, which is not a valid assumption in the context of IoT, especially on resource-constrained devices.

Another extension to payment channels is the idea of a payment hub to use the nodes that have multiple open channels (hub) to circulate the payments. For example, Tumblebit [69] extends the payments with anonymity making them unlinkable by using an untrusted intermediary. However, the involvement of the intermediary for each payment leads to performance issues. Perun [70] proposes the virtual payment hub to avoid this problem. NOCUST [71] proposes the separation of the functionality of the payment hub into two components. First, an off-chain operator server handles every transfer. Second, an on-chain smart contract verifies the payments. Finally, Ye et al. [38] 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.

Similar to payment channels, another proposal to scale the payment system of blockchain is sidechain [72]. *Sidechain* is a smaller, lighter parallel structure similar to blockchain but with a shorter living period. The protocol includes an on-chain smart contract that bridges several lighter and faster sidechains. The nodes can exchange the off-chain tokens that have a correspondence in the main blockchain. The Plasma [33] framework is the proposal of the Ethereum

team to scale the blockchain network. However, current sidechains do not take into consideration the challenges of low-power IoT devices.

Authentication and distributed storage of IoT data. In parallel to our work of IoTLogBlock (presented in Chapter 3), Black-Box IoT [73] proposes a blockchain-based storage system for IoT devices. With our work, we introduced an offline recording architecture of IoT transactions by utilizing the Hyperledger Fabric blockchain and an optimistic contract signing protocol. Black-Box IoT aims to provide authentication and data storage using Hyperledger Fabric without providing a fair exchange protocol (explained earlier). In this way, Black-Box IoT provides a lightweight approach by utilizing a hash-based digital signature. Black-Box IoT makes stronger assumptions regarding the trust and interaction among the IoT devices. With IoTLogBlock, we utilize a contract signing protocol [74] that allows two or more parties that do not trust each other to sign a pre-defined agreement. The protocols provide fairness, timeliness, and an abuse-free property [75].

Multi-signature for micro-payments on IoT Devices. Similar to our work with TinyEVM (presented in Chapter 5), Bolt [76] proposes a 3-of-3 multi-signature system to allow IoT devices to participate in the lighting network of Bitcoin. They introduce a protocol to create a 3-of-3 multi-signature channel where three parties are involved: the IoT devices, Lightning Network (LN) gateway, and LN bridge. With their protocol, IoT devices can utilize the lightning network to perform their transaction without constant communication. However, their protocol does not take into consideration the local context of the IoT devices (e.g., sensor data), and they do not provide support for automated agreements (i.e., smart contracts). With TinyEVM, we need a 2-of-2 signature to create a channel between two devices, and the devices can execute off-chain smart contracts with specific IoT-opcodes to utilize the local interaction of sensors and actuators inside a smart contract. Finally, Bolt targets powerful IoT devices such as a Raspberry Pi with a 1.5 GHz multicore CPU and GBs of RAM, while TinyEVM targets devices with 32-64Hz of CPU and a couple of MBs of RAM.

1.4.2 Deep Learning on Embedded Systems

Even though machine learning is mainly associated with cloud services, we see aspects of machine learning gradually integrated into IoT systems [9–11]. Machine learning on IoT enables anomaly detection [12], human activity recognition [13], computer vision [14], and keyword spotting [15], but has demanding performance, memory and energy requirements. So far, the main parts of training, optimization, and compression of neural networks are commonly conducted in the cloud. IoT devices are running neural networks for inference-only tasks [9–11]. Typical methods for compressing neural networks for IoT devices include: pruning, network architecture search NAS [77], and quantization [78]. We primarily focus on 8-bit quantized methods due to the wide support and success on low-power devices and keeping the accuracy close to the original networks. Even though pruning and partitioning can significantly reduce the network's size, they suffer from accuracy drops. Other methods for quantizing neural networks are feasible, for example, with half-wave Gaussian quantization [78] and low-bit neural networks [48,79], but they suffer from high accuracy

losses.

Transfer learning. IoT devices are deployed and interact in real-world environments, and their initial problem can change [22]. Transfer Learning (TL) [51, 80, 81] can allow adapting an already-trained network to a related problem without needing extensive training data and computing resources. For this, TL utilizes pre-trained deep neural networks and re-trains parts of the network using a smaller dataset [51, 80]. TinyTL [82] and DeepCham [83] have shown the feasibility of transfer learning on edge devices (e.g., Raspberry Pi). However, they overlook the challenges of low-power devices, especially the energy and memory limitations, and they do not consider quantized pre-trained networks. In a similar direction, quantization and knowledge distillation (QKD) [84] on edge devices is a method for using a high-precision model to create quantized networks for other domains. Our method differs by dynamically adapting the domain shift for using an existing model without the need to download a new model from on edge/cloud services. Other methods in edge computing divide the task of transfer learning to multiple devices [80]. They improve the performance and maximize the overall precision of machine learning. However, they suffer from data movement to the cloud, which would swiftly drain the battery of a typical low-power device.

Distributed learning. Another way to overcome the centralized data collection and processing of common deep learning techniques is distributed and Federated Learning (FL). These distributed collaborative approaches allow participating devices to train a common global model without data sharing. For example, FedHome [85] and FedHealth [3] use edge devices to collect and train a common model from IoT devices without sharing them to the cloud. However, training methods of federated algorithms often use Homomorphic Encryption (HE) [86] or Secure Multiparty Computation (SMC) [87] to protect the privacy of the users. Due to this, federated learning today is commonly done on edge class devices. These requirements are ill-fitting for low-power devices with limited resources.

Offloading. Alternative methods without focusing on learning capabilities on the device look into the partitions of networks between an IoT and edge, minimize execution latency [88], and keep a reasonable communication cost. Similarly, distributed inference hierarchies can offload inputs between cloud, edge, and IoT devices [89]. Adapting these methods to shift the problem domains is quite challenging as several updates need to be broadcast through the network.

1.4.3 Firmware Verification

Another relevant part is the essential techniques of the broader field of secure boot. The threat space for attacks on the firmware of IoT devices includes Rootkits [90] and Bootkits attacking the boot process and devices' firmware [91]. With secure boot, the goal is to ensure that firmware during the boot process code is verified to come from a trusted source and has not been compromised by an attacker. Kushwaha [92] presents an approach to secure-boot for PC systems based on a USB key containing and verifying most of the boot process. This technique can easily be adapted to embedded systems. Dietrich and Winter [93] describe a software Mobile Trusted Module (MTM) which starts at

a later point in the boot process (after the initialization process). The authors propose a backward verification of the already executed stages of the boot process whenever MTM is active and running.

Recent works demonstrate the need for securing the boot process of connected devices from consumer level printers to industrial robots [94, 95]. There is a large body of research in the field of securing and verifying the boot process. Khalid et al. [96] discuss the difference between secure and trusted boot. Liu et al. give a slightly different approach for system verification [97] and Lebedev et al. [98] give examples of a remotely attested system using embedded systems. For recent work regarding IoT devices, Asokan et al. [99] focus on solutions regarding the firmware update on large-scale IoT deployments. For constraint devices, Boot-IoT [100] proposes an authentication scheme for secure bootstrapping.

1.5 Research Questions

This thesis targets software architectures and algorithms for data provenance, automation, and machine learning that conventionally run on powerful high-end devices. We redesign and tailor these architectures to low-power IoT, including the implementation and evaluation on real test-beds. We identify key research questions and challenges in providing IoT systems capabilities of blockchain, machine learning, and trust on the software that are typically found on powerful cloud nodes. The studies of these systems examine inquiries about the feasibility of IoT applications to incorporate local data and processing on low-power IoT devices. Specifically, this thesis identifies and answers the following four Research Questions (RQs).

RQ1: How can we redesign and bring algorithms traditionally run on powerful reliable nodes to low-power IoT devices?

RQ2: How can we trust and verify transaction records formed by IoT sensors and actuators in open environments?

RQ3: What are the main trade-offs between IoT device processing versus offloading to the cloud?

RQ4: How can we tailor learning algorithms to low-power devices and benefit from local data?

RQ1 reflects the core of this thesis as it relates to the essential problem of enabling capabilities on resource-constrained devices fundamentally needed for the next generation of IoT applications. The next generation of IoT applications require on-device machine learning, automated decision making, and data provenance by bringing functionality commonly found on cloud services to the IoT devices.

RQ2 occurs from the observation that the interaction of numerous IoT will produce millions of transactions coming from low-cost embedded devices. The ability to store millions 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. Moreover, IoT applications need to include sensor readings and actuators as part of IoT transactions.

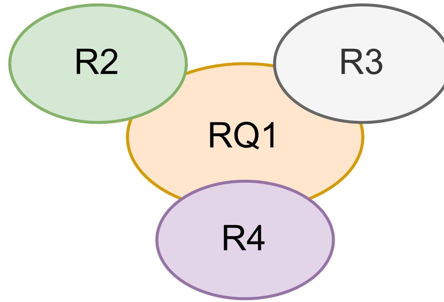


Figure 1.10: The magnitude of each research question in connection to the thesis. RQ1 is at the core of the thesis and the other questions explore side directions.

Table 1.1: Research questions and the corresponding chapters that address them

	Part II			Part III		
	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7
RQ1	●	●	○	○	●	●
RQ2	●	●	●	○	○	○
RQ3	●	●	○	○	●	●
RQ4	○	○	○	●	●	●

RQ3 is relevant in the context of deciding between local processing and offloading to the cloud. By doing local processing, algorithms intended to operate on powerful cloud nodes are now deployed on resource-constrained devices. This new design raises research questions regarding the trade-offs in the performance and optimization of IoT devices.

RQ4 approaches a significant optimization opportunity missed by the current cloud paradigm: *on-device learning specialized for locally collected IoT data*. We investigate the feasibility and challenges of utilizing dynamically on-device learning on resource-constrained devices. Efforts for tailoring deep learning methods on low-power IoT devices face significant challenges as training neural networks requires large datasets, unaffordable for low-power devices with severe memory constraints. Deep neural networks have millions of trainable parameters, and IoT devices lack the high-end CPU performance to train an entire neural network from scratch. Moreover, low-power IoT devices depend on batteries where deep learning algorithms would need to incorporate duty cycling.

1.6 Thesis Contributions

In this section, we outline the contributions of the appended papers in relation to our research questions. Table 1.1 summarizes all the research questions with the corresponding chapters that address them. Next, we summarize each paper of the thesis, and its contributions.

1.6.1 Blockchain and Smart Contracts on Verified IoT Devices (Part II)

In Part II of the thesis, we target offline recording of IoT transactions with smart contracts, and device software integrity with secure boot. The common issue we address is the trust in IoT devices and the data they produce. We contribute towards **RQ 1, 2, and 3**, showing: i) We can verify IoT records of transactions without constant cloud communication. ii) We can include sensor readings and actuation of the device as part of automated transactions. iii) We can verify the firmware of the device during the boot process with a marginal cost.

Chapter 2: Recording Off-line IoT Transactions (Paper A)

In Chapter 2, 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, as the devices are owned and operated by different entities. We investigate the challenges of managing and recording transactions of IoT devices using a blockchain network. Through this work, we contribute towards **RQ 1, 2, and 3**, by designing and implementing an architecture tailored to low-power devices. We show the trade-offs of using a blockchain to record transactions coming from resource-constrained devices.

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 online entity to report any misbehavior. We design a smart contract to act as the online entity to report complete transactions or misbehaving nodes. The blockchain network is used to store the transactions permanently.

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 face a conflict of interest and do not necessarily trust each other. 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. Our design took care of the above threats and motivated our prevention measures.

We evaluate our design on resource-constrained devices like TI CC2538 [101] 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 resource-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.

Chapter 3: Smart Contracts on Low-Power IoT Devices (Paper B)

In Chapter 3, 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 1, 2, and 3**, 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. This scenario shows 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 [101], and it consists of three components: a) publishing the on-chain smart contract, b) creating off-chain channels, and c) committing to the on-chain part. 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 the Ethereum Virtual Machine with IoT opcodes to execute smart contracts on resource-constrained devices. We investigate the trade-offs to execute smart contracts on resource-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 resource-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.

Chapter 4: Performance of Secure Boot in Embedded Systems (Paper C)

In Chapter 4, 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 the capability to perform 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 2**, 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 widespread 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 Embedded Deep Learning on IoT Devices(Part III)

The third part (III) of the thesis targets deep learning on low-power IoT devices to benefit from local data. We study architectures and system design by tailoring learning and inference algorithms from high-end devices to lower-power IoT devices. We contribute towards **RQ 1, 3, and 4**, demonstrating: i) We can balance the trade-offs of deep learning on low-power IoT devices. ii) We can adapt to changes in the problem domain of deep neural networks. iii) We can adjust and retrain deployed deep neural networks on the device.

Chapter 5: Performance of Deep Neural Networks on IoT Devices (Paper D)

In Chapter 5, we design a benchmark for evaluating various frameworks for deep learning on IoT devices. Even though advances in deep learning have revolutionized machine learning by solving complex tasks such as image, speech, and text recognition, the training and inference of deep neural networks are quite resource-intensive. Recently, researchers made efforts to bring inference to IoT edge and sensor devices which have become the prime data sources nowadays. However, running deep neural networks on low-power IoT devices is challenging due to their resource constraints in memory, compute power, and energy. In this Chapter, we present a benchmark to grasp these trade-offs by evaluating three representative deep learning frameworks: uTensor, TF-Lite-Micro, and CMSIS-NN. In this way, we contribute towards **RQ 4**, by revealing key trade-offs of deep neural networks on low-power devices. Our benchmark reveals significant differences and trade-offs for each framework and its toolchain: (1) We find that uTensor is the most straightforward framework to use, followed by TF-Micro, and then CMSIS-NN. (2) Our evaluation shows significant differences in energy, RAM, and Flash footprints. CMSIS-NN is the most efficient in terms of energy, followed by TF-Micro and then uTensor, each with a significant gap.

Chapter 6: Transfer Learning on IoT Devices (Paper E)

In Chapter 6, we propose **MicroTL** to enable transfer learning using quantized models on resource-constrained devices. Deep neural networks are becoming readily available on IoT devices (e.g., health trackers, smart cameras, humidity sensors), but current approaches depend on enormous collections of sensor data in the edge and cloud services, where all steps happen: training, optimization, and compression. However, personalized data from IoT sensors are typically available after deployment, with limited accessibility due to energy-preserving communication and privacy concerns. IoT devices with inference-only capabilities cannot adapt to environmental changes, particularly when the domain problem shifts. Train a completely new model on-device is impractical considering a typically low-power device is running on 32-64 MHz with a couple of KBs of RAM, operating on batteries. In this way, we contribute towards **RQ 1, 3, and 4**, by tailoring on-device learning to low-power devices, identifying key trade-offs, and incorporating local data.

MicroTL's main objective is to adapt deep neural networks to end-user preferences without the communication dependency on sending sensor data from IoT devices to the cloud and neural network updates from the cloud to IoT devices. MicroTL allows training fully connected layers personalized to the end-user, and it keeps the data on the device, avoiding sending sensitive or private data to the cloud. MicroTL assumes a pre-initialization step of downloading or flashing a pre-trained neural network on IoT devices. MicroTL is based on two key observations. First, we can reduce the training memory requirements, by dynamically collecting intermittent outputs of hidden layers in integer precision and converting to floating-point precision only during training. This approach provides a sufficient dynamic range for calculating gradients during back-propagation, avoiding duplicated weights. Second, training fully connected layers in floating-point precision while keeping the pre-trained network in integer format reduces the demand for computational and energy resources without affecting the overall transfer learning process.

Bringing TL to low-power devices, faces trade-offs in computational, energy, and memory constraints. By introducing MicroTL, we bring transfer learning into low-power IoT devices allowing for changes in the problem domain. We implement and evaluate MicroTL on low-power devices. Notably, we found that MicroTL takes 3x less energy and 2.8x less time than transmitting all data to train a completely new model on the cloud, showing that it is more efficient to adopt an existing pre-trained network on the IoT device rather than dutifully communicating with the edge or cloud to create a new one.

Chapter 7: On-device Learning for IoT Devices (Paper F)

In Chapter 7, we design **MiniLearn**, an on-device learning system to benefit from local data. As inference with deep neural networks on low-power IoT devices (i.e., equipped with a few hundred KBs of RAM) has become readily available several new opportunities emerge. However, the operational device parameters may change after deployment (e.g., memory and energy demands), and deployed neural networks have a hard time adapting, such as reducing network size and dynamically adding or removing classes. There is a

significant optimization opportunity missed: *on-device learning specialized for locally collected real-time IoT data.* In this way, we contribute towards **RQ 1, 3, and 4**, by tailoring on-device learning to low-power devices, identifying key trade-offs, and incorporating local data.

Resource-constrained devices lack computational, memory, and energy resources for training neural networks, which is thus typically conducted on powerful cloud services. MiniLearn takes on the above challenges, and it demonstrates the existence of further opportunities by dynamically utilizing on-device learning on low-power IoT devices. MiniLearn retrains and optimizes pre-trained quantized networks for a subset of the initial classification tasks, using IoT data collected during deployment in real-time. The main objective is to provide on-device learning specialized for the user using locally collected IoT data. MiniLearn retrains neural networks to end-user preferences without communicating with the cloud by keeping the data on the device and preserving the user's privacy.

MiniLearn is based on two key observations. First, we can reduce the training parameters by dynamically pruning the hidden layers in integer precision and converting them to floating-point precision during training. Second, we can fine-tune the fully connected layers in floating-point precision while converting other hidden layers back into integer format and reducing the demand for computational and energy resources without affecting the learning process. Evaluation results on embedded hardware demonstrate that after MiniLearn, the network can take up to 2x less inference time and 3x less memory than the original network.

1.7 Conclusion and Future Directions

In this thesis, we study the next generation of the Internet of Things and the progress from simple wireless sensors and actuators to complex systems with capabilities such as machine learning and automated transaction processes. These capabilities depend on centralized cloud architectures demanding significant communication between the devices and the cloud. The success of the next generation of IoT systems depends on automating transactions and learning from the local environment with reduced cloud communication. Thus, the thesis focus on how to enhance IoT devices with self-reliance by targeting architectures and algorithms for data provenance, automation, and learning that are conventionally running on powerful high-end devices.

We examine key research questions regarding the trust and feasibility of applications incorporating local data in learning and automated agreements on low-power IoT devices. We propose, implement, and evaluate these designs on resource-constrained IoT devices with experimental studies. The thesis includes six chapters answering the key research questions in two parts: a) establishing trust and enabling automated transactions for IoT devices in a transparent way with smart contracts and the support of blockchain services. b) facilitating machine learning and inference on low-power devices with limited cloud communication.

Finally, based on our work's key insights, we identify two emerging directions for future work. First, the append-only nature of the blockchain raises

questions about the scalabitily of the current data storage. The current solution involves heavy operations and complicated architectures. Promising research directions include scalable and lightweight designs for transaction automation. For example, off-chain payments and extensions of the current design of smart contracts can improve the throughput of IoT transactions. Second, solving the communication and performance issues of distributed algorithms for training algorithms on low-power devices can improve the accuracy and privacy of the users. For example, enabling federated learning on low-power devices can give current architectures access to an enormous amount of data.

Part II

Blockchains and Smart Contracts on Verified IoT Devices

Paper A

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.

CHAPTER 2

IoTLogBlock: Recording Off-line IoT Transactions

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 Chapter, 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.

2.1 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

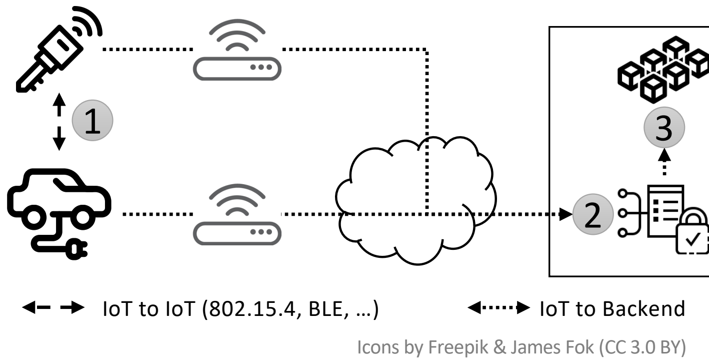


Figure 2.1: Motivating scenario: car rentals where (1) a smart-key and a smart-car perform off-line transactions using a contract signing protocol over low-power wireless technologies. Later, the IoT devices independently forward their transactions via an edge device to the blockchain where (2) a smart contract validates and then (3) stores the transactions immutably.

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 [5] [6], 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 [57] [31] 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 [102] 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 2.1. The contract signing protocol

allows two or more nodes to sign an off-line transaction mutually and achieve non-repudiation [103]. 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 Chapter is organized as follows. In Section 2.2, we provide the necessary background regarding our approach. In Section 2.3, we describe a motivating example and our adversary model. In Section 2.4, we provide the system design and highlight security aspects. We evaluate our results in Section 2.5, which is followed by a discussion and a description of related work before concluding this Chapter.

2.2 Overview and Background

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

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

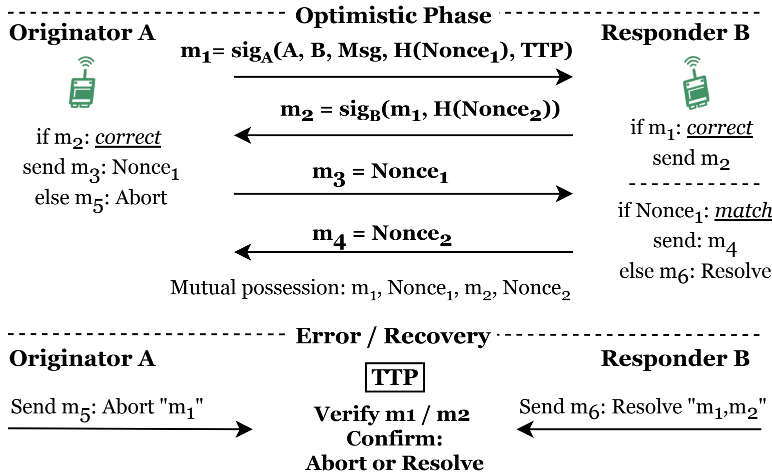


Figure 2.2: Signature exchange in the Asokan-Shoup-Waidner protocol. In the optimistic phase, two nodes agree to sign a contract. At the end of the protocol, both participants have a copy of the mutually signed agreement. In the error/recovery phase, a node tries to abort or resolve a previous round of the protocol using a Trusted Third Party (TTP).

2.2.1 Contract Signing Protocols

Contract signing protocols [74] 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 [75]. Two parties (Alice and Bob) exchange their signatures in a fair-way [104], where Alice can obtain Bob's signature only if Bob can obtain Alice's signature and vice-versa. Timeliness [75] means that a participant is not able to make the other wait for an indefinite amount of time. Finally, with the abuse-free property [75], 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) [105], protocols without TTP (e.g., time commitments [106]), and protocols with an off-line TTP (e.g., optimistic [104]). 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)** [104] protocol, which provides fairness and timeliness, but it is not abuse-free [75]. The ASW protocol allows two parties to exchange signatures as shown in Figure 2.2, 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).

2.2.2 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 [56], 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.

2.2.3 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 [26], 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 [31], which is a collaboration between Linux Foundation and IBM to provide an open-source distributed ledger. The platform has three main components (see Figure 2.3). First, a *Membership Service Provider (MSP)* manages the identities on the permission-based blockchain [31]. 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.

2.3 Application Scenario and Adversary Model

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

2.3.1 Application Scenario

As a motivating scenario, we consider a car rental service with regular customers [107]. 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

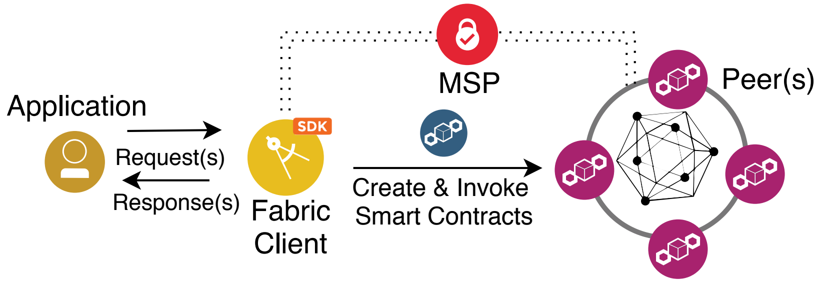


Figure 2.3: The Hyperledger Fabric consists of: 1) the Membership Service Provider (MSP), which issues the access list of participants, 2) the private network of peer(s) that maintain the blockchain, and 3) the Fabric client which provides the application interface.

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.

2.3.2 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.

2.4 System Design

2.4.1 Design Overview

There are three key building blocks to IoTLogBlock (Figure 2.1): (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 2.1. These steps will then be further elaborated below.

2.4.1.1 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 [108], 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 [109], with an optimistic two-party contract signing protocol, which is further described in Section 2.4.3.

2.4.1.2 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 2.1). This step builds on established mechanisms and is not further described in the design of IoTLogBlock.

2.4.1.3 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 2.4.4 and the final step, permanent storage, is described in Section 2.4.5. We conclude in Section 2.4.6 with the security analysis of potentially misbehaving nodes.

2.4.2 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

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
```

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.

2.4.3 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 3.2). 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.

2.4.4 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 [104], 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 2.2). 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, 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 2.2) to the smart contract. The resolve sub-protocol is also invoked when a node tries to abort even if it has already sent $message_2$.

2.4.5 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 3.2.

2.4.6 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.

2.5 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 [31].

Goals. With this section, we answer the following questions: a) is IoTLog-Block 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.

2.5.1 Implementation and Experimental Setup

IoT Software Stack. We implement the contract signing protocol in C as an application for Contiki-NG [110]. 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 [108] 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 [111], 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]

2.5.2 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 2.3) we rely on the CC2538 data sheet [101] and a previous evaluation [112].

Memory Footprint. Table 2.1 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.

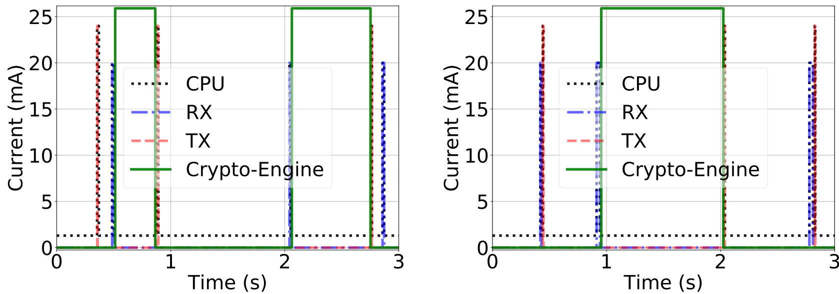
Performance. Next, we evaluate the performance of the cryptographic functions. All cryptographic operations are performed by the cryptographic engine running at 250 MHz, and in Table 2.2 we present the performance of each task. The average time to complete all cryptographic functions of the contract 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 2.4) gives us a total of 3 s per transaction. We argue that this overhead, 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 [108]. In Figure 2.4, 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.

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

Table 2.1: Memory Footprint of the IoT application (considering max size) on CC2538, which facilitates 32KB of RAM and 512 KB of ROM.

Component	**RAM** Bytes	/ Percent	**ROM** Bytes	/ 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%



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

(b) Responder: This node responds accordingly.

Figure 2.4: The electric current drawn by a complete round of the contract signing protocol.

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 2.4. Next, the originator starts the protocol by signing the first message, which takes 0.3 s (see Table 2.2). 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.

In Table 2.3, 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

Table 2.2: Performance of cryptographic functions using the CC2538 cryptographic engine running at 250 MHz.

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 2.3: 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) [101].

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

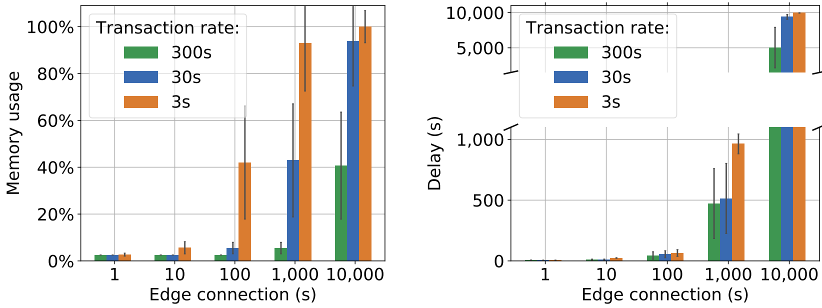
capacity, i.e., 2000 mAh [113]. As a result, we can expect 10,000 Joules of energy from the cells, which allows IoTLogBlock to perform roughly in the order of 100,000 transactions. While this is merely an estimate, we argue that this order of magnitude of transactions is practical for a wide range of application scenarios, including ours of a battery-powered smart key.

Reflecting on previous work [112] [114], 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.

2.5.3 System Performance of IoTLogBlock.

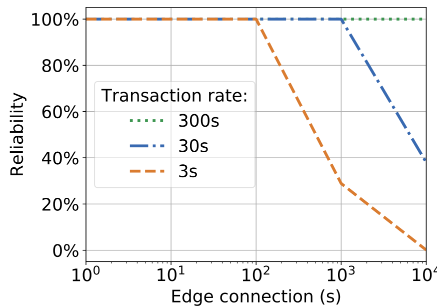
We evaluate the overall system in terms of a) local storage capacity b) delay of registering a transaction to the blockchain, and c) the reliability of nodes when creating off-line transactions. In the experiments, we utilize the remaining available RAM for storing off-line transactions while waiting 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



(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 2.5: 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.

previously, a transaction takes 3 s. Thus, in our evaluation, two nodes generate transactions every 3, 30, and 300 s (see Figure 2.5).

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 2.5a 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 2.5c and discussion below).

Register Delay. In Figure 2.5b, 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 2.5c. We notice that the reliability highly depends on two factors: the periodicity of the edge connection, and the transaction generation rate.

2.6 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 3.4). 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 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.

2.7 Related Work

Fair exchange. Similar to our work, FairSwap [61] 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 [5] [6] 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 [53] of integrating IoT with blockchains. There are approaches to integrate IoT devices with blockchains such as AGasP [56] and Edgechain [57]. 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 [115] [114]. IoTLogBlock differs from previous work in that we use blockchain to record transactions and a contract signing protocol to achieve non-repudiation.

2.8 Conclusion

In this Chapter, 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.

2.9 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 B

Christos Profentzas, Magnus Almgren, Olaf Landsiedel

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

Proceedings of the 40th IEEE International Conference on Distributed Computing Systems (ICDCS) 2020

CHAPTER 3

TinyEVM: Smart Contracts on Low-Power IoT Devices

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 Chapter, 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.

3.1 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 [23, 116]. 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 [27], 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 [24, 117], 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 Chapter with the following structure. In Section 3.2, we provide the necessary background for TinyEVM. In Section 3.3, we provide an overview of our application scenario and the design requirements. In Section 3.4, we provide the system design of TinyEVM. In Section 3.5, we provide a security analysis of our design. We present the evaluation results in Section 3.6. Finally, we provide the related work in Section 3.7 and the conclusion in Sections 3.8.

3.2 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.

3.2.1 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 [118,119]. The research community responded with several proposals to scale the blockchains like sharding [34,35], consensus algorithm variations [25], and trusted execution [24]. In this chapter we focus on three prominent proposals: (a) side-chains [33], (b) payment channels [36,37], and (c) payment networks [38], which we further described below.

3.2.2 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 [27] 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 [117]. 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.

3.2.3 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) [36,37].

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 [36, 38] reuse existing user channels to route payments off-chain. Second, state channels [63, 69, 120] 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.

3.2.4 Side-Chains

Another proposal to scale blockchains is the idea of side-chains [33, 121]. 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.

3.3 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.

3.3.1 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 3.1. 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

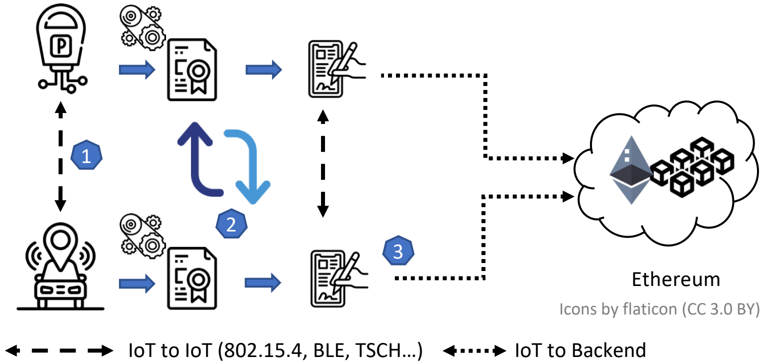


Figure 3.1: The parking application scenario: 1) The vehicle and the parking lot communicate via a short-range protocol. 2) They open an off-chain payments channel with an initial deposit and perform offline payments. 3) A node can at anytime submit a final state to the blockchain, in our case Ethereum.

payments or updates on the payment rates based on the time of day. At the 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.

3.3.2 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.

3.3.3 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.

3.3.4 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.

3.4 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.

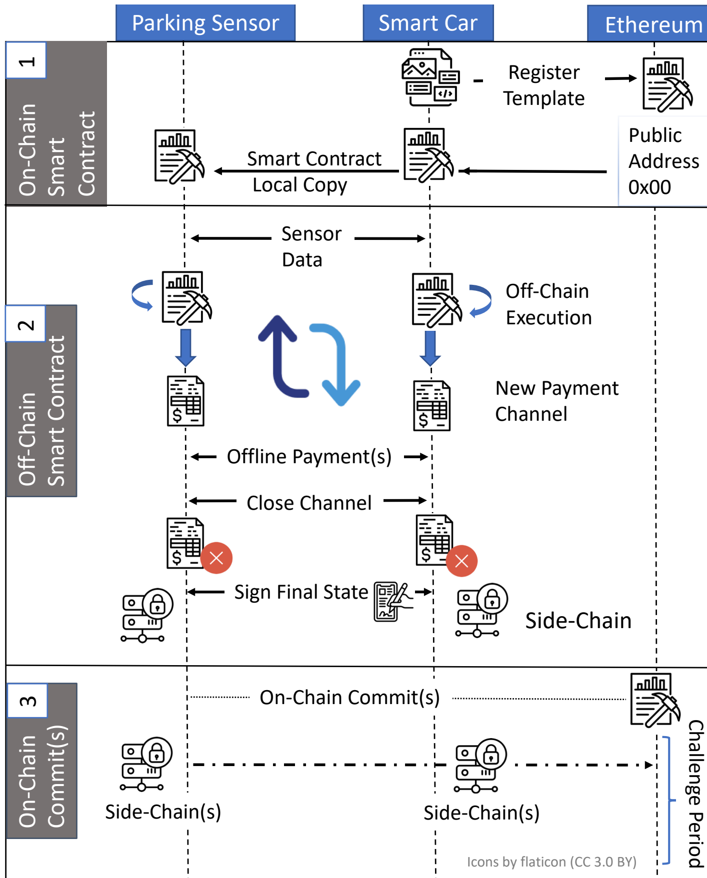


Figure 3.2: The system includes three phases: 1) Publishing the on-chain template smart contract. 2) Creating an off-chain payment channel and signing (multiple) payments. 3) On-chain commit of a final state, which activates the challenge period, and nodes can dispute with their local side-chains.

3.4.1 On- and Off-Chain Transactions: Three Phases

The deployment and execution of smart contracts include three high-level phases, visible in Figure 3.2.

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

Table 3.1: 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.

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

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.

3.4.2 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 3.1, 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.

3.4.3 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 [122].

The template is a factory-smart-contract [123], which can create and deploy child contracts dynamically, see Listing 3.1. The child contracts in our scenario are the payment channels, see Listing 3.2.

3.4.4 Off-Chain Smart Contract

The second phase in Figure 3.2 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 [108] or BLE [124]. 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

```
1 contract Template {
2   address[] PaymentChannels;
3   uint Balance;
4   address payable public Receiver;
5   uint Logical-Clock = 0;
6   MerkleSumTree Side-Chain-Root;
7
8   function CreatePaymentChannel (uint64 Money) public  {
9       newPaymentChannel = new PaymentChannel( receiver, Money);
10      PaymentChannels.push(newPaymentChannel);
11      Logical-Clock += 1;
12  }
13  function OnChainCommit{...} //user specific
14  function Challenge{...} //user specific
15 }
```

Listing 3.1 : Factory Template in Solidity

```
1 contract PaymentChannel {
2  address payable public Sender;
3  address payable public Receiver;
4  uint public sensor_data;
5
6 constructor(address payable _recipient) public payable {
7   sender = msg.sender;
8   recipient = _recipient;
9   assembly{
10      0x0c //IoT sensor opcode
11      sstore(0x0c) // Store sensor data
12  }
13 }
14 function close(uint amount, bytes memory signature) public
      payable {
15  require(msg.sender == recipient);
16  require(isValidSignature(amount, signature));
17  recipient.transfer(amount);
18  selfdestruct(sender);
19 }
20}
```

Listing 3.2 : 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.

3.4.5 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 [33], 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.

3.5 Security Analysis

Similar to other off-chain systems [33, 72], 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) [33].

3.6 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.

3.6.1 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 3.1.

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 [101]. 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.

3.6.2 Ethereum Virtual Machine on IoT Devices

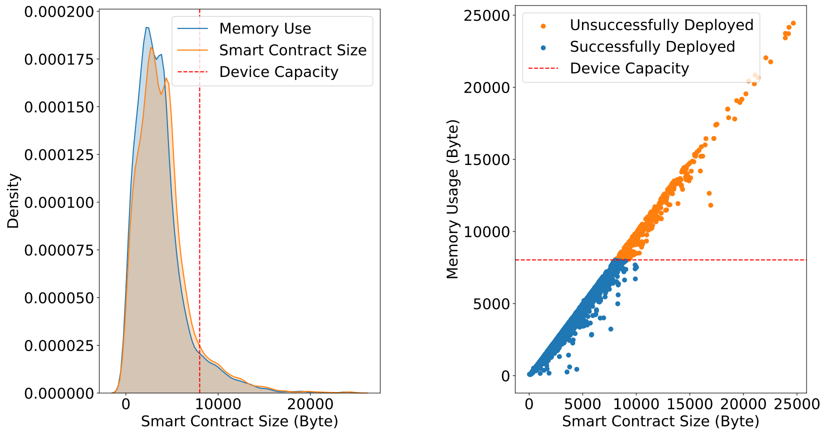
We evaluate the resource efficiency of TinyEVM and its ability to deploy off-the-shelf smart contracts. For this, we collect roughly 7,000 publicly available smart contracts to test our platform. The smart contracts are verified by Etherscan.io, a widely used blockchain explorer.

3.6.2.1 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.

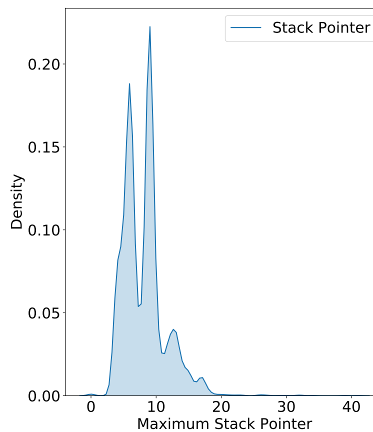
In Figure 3.3a, we see the distribution of the smart contract size and the memory usage in TinyEVM after the deployment. We observe that the average size of a smart contract is 4 KB (see also Table 3.2). The maximum size is 25 KB and the minimum size is 28 Bytes. The sizes of the smart contracts are 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 3.3, 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 3.3b, we observe the memory usage



(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 3.3: 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.

needed to deploy the smart contracts. We notice the positive correlation between memory use and the size of the smart contract. However, the final 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.

We continue our analysis with the virtual machine's stack usage during the experiments. In Figure 3.3c, 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 3.2, 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.

3.6.2.2 Deployment Execution Time

In Figure 3.4, 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 3.2 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.

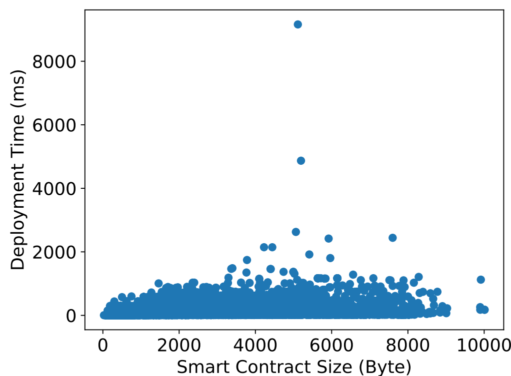


Figure 3.4: The time of deploying a smart contract in relation to its bytecode size. The average time is 215 ms, but we can notice there are some outliers.

Table 3.2: An overview of memory and deployment time of the 5,953 successfully deployed smart contracts.

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

	RAM		**ROM**	
Component	**Bytes**	**Percent**	**Bytes**	**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%

3.6.3 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 [125] that has a 30-microsecond resolution timer.

3.6.3.1 Memory Requirements.

We present the memory foot-print of the payment channel in Table 3.3 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.

Table 3.4: 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) [101], when not active.

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

3.6.3.2 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 3.5, 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.

3.6.3.3 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 [108], 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 3.5, 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.

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 3.5. 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

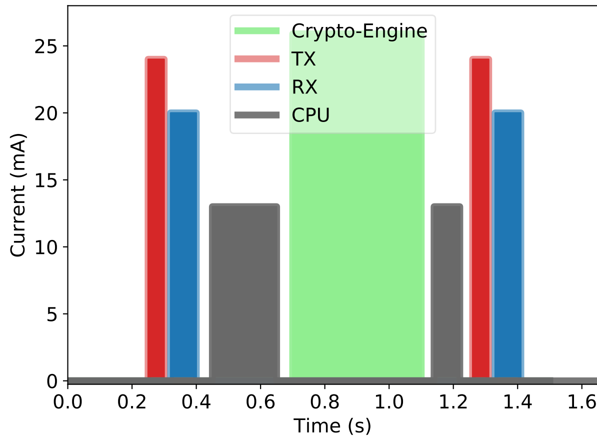


Figure 3.5: The electric current (in mA) drawn by a complete round of the off-chain payments channel. There are three discrete pieces involved: the wireless communication, the virtual machine execution, and the cryptographic engine.

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 3.4, 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 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.

3.7 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

Table 3.5: Performance of cryptographic operations. Keccak256 is implement on software, the other operations are using the CC2538 cryptographic engine running at 250 MHz.

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

blockchain for IoT applications. IoTLogBlock [116] 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 [23] 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 [64] was one of the first implementations of such networks for Bitcoin. The equivalent network for Ethereum is Raiden [65].

There are three main challenges to make PCs usable in practice. One challenge is to open a PC in both directions. Duplex micropayments [66] 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 [36] 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 [37] 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 [67], and SpeedyMurmurs [68]. 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 [69] 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 [70] proposes the virtual payment hub to avoid this problem. Third, payment hubs can lead to collateral fragmentation. NOCUST [71] proposes the separation of the functionality of the payment hub to two components. First, an off-chain operator server han-

dles every transfer. Second, an on-chain smart contract verifies the payments. Finally, Ye et al. [38] 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 [72] 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 [33] 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 [24] provides a bridge between HTTPS-enabled data websites and the Ethereum blockchain. Moudoud et al. [117] 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 [126–128] 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 [129], however, these solutions are not designed for resource-constrained devices.

3.8 Conclusion

In this Chapter, 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.

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

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.

CHAPTER 4

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 Chapter 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%.

4.1 Introduction

The Internet of Things (IoT) will bring connectivity to everyday objects and devices, including vehicles [1], autonomous robots [12], and smart home appliances [2] (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 [130], data centers [8] and also portable devices such as smartphones or tablets [131]. 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 [132]. 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 [95]. 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 [133]. 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 chapter focuses on securing the boot process of medium-scale microcontrollers (e.g., Raspberry Pi, Beaglebone) equipped with an embedded operating system.

The chapter 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, 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 chapter is structured as follows. First, Section 4.2 introduces basic concepts and definitions of secure boot. Next, Section 4.3 presents our threat model. We present secure boot on off-the-shelf embedded

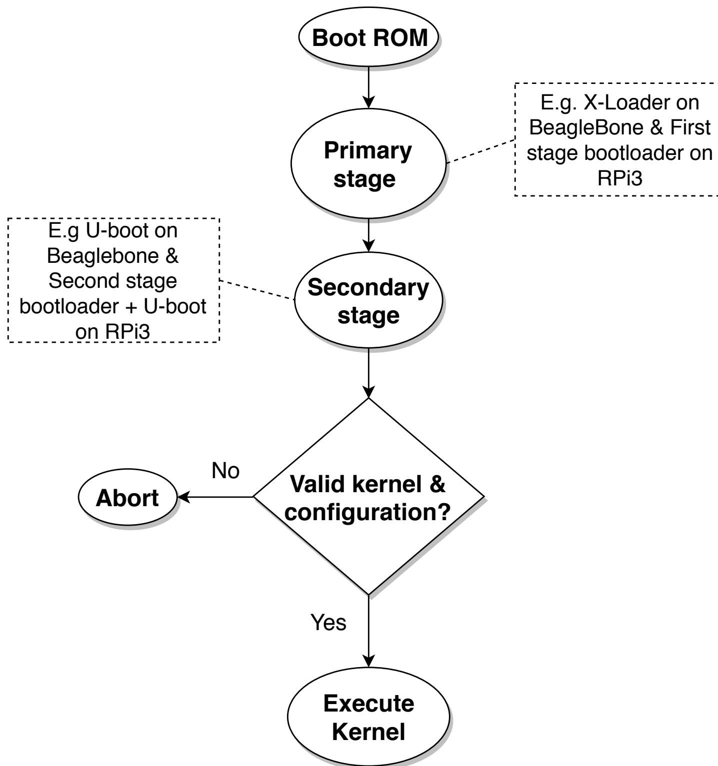


Figure 4.1: The boot process of the medium-scale embedded systems consists of multiple stages. The Boot ROM and first stage boot-loader are hard-coded by the manufacturer and are hard to modify. The second stage and the kernel code are modifiable and usually stored in flash memory.

hardware in Section 4.4 and evaluate its performance in Section 4.5. In Section 4.6 we discuss the limitation of secure boot in IoT devices. Section 4.2 presents related work and Section 4.7 concludes the chapter.

4.2 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 [134] into several stages (see Fig. 4.1). 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) [135]. The Trusted Computing Group (TCG) [135] 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. 4.1), 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 [8, 93, 136]. Especially, *Secure Boot* has been among the standard techniques to define a secured process to assure the integrity of each booting steps [137]. In table 4.1 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 [94, 95]. There is a large body of research in the field of securing and verifying the boot process. Khalid et al. [96] 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. [97] and Lebedev et al. [98], 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 [136]. For recent work regarding IoT devices, Asokan et al. [99] focus on solutions regarding the firmware update on large-scale IoT deployments. For constraint devices, Boot-IoT [100] propose an authentication scheme towards secure bootstrapping.

4.3 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

Table 4.1: Terminology comparison

Term	Halt	RoT	Verification	Added HW
Secure boot [96]	Auto-termination	Boot ROM	By certificate authorities (Remote attestation)	Not specified
Trusted boot [8]	Letting users decide	Boot ROM	Compare hash values	HSM
Verified boot [136]	Letting users decide	Boot ROM	Stored cryptographic hash comparison	Not specified
Measured boot [138]	No termination	BIOS	Measures hash of objects and logs them	Not specified

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 4.2. 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 [139]. 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 [140], the OS [141] and also the boot process [94]. 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) [142] is a promising solution. In our threat model and further system design, we focus on the new attack vector that Internet connectivity brings.

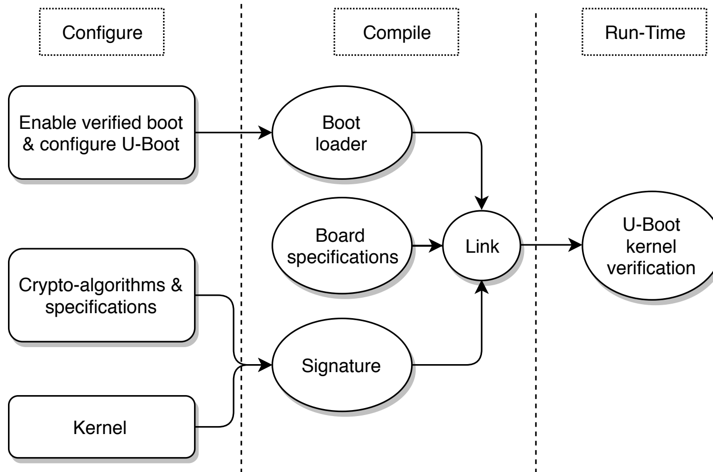


Figure 4.2: From U-Boot configuration to deployment: First, we configure the U-Boot to include the verification module. Next, we link the object files to produce the secure version of U-Boot. Finally, we deploy the executable file on the platform.

4.4 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 [8, 96, 136] in the field, and we do not claim their novelty. Instead, the contribution of this chapter 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.

4.4.1 Software-Based Secure Boot with U-Boot

For the software-based method, we rely on the U-Boot [143] 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* and *Run-Time*, see Figure 4.2. The software-based secure boot process extends the boot process with an additional verification step, see Figure 4.3. As a result, the bootloader only boots the operating system once it has successfully

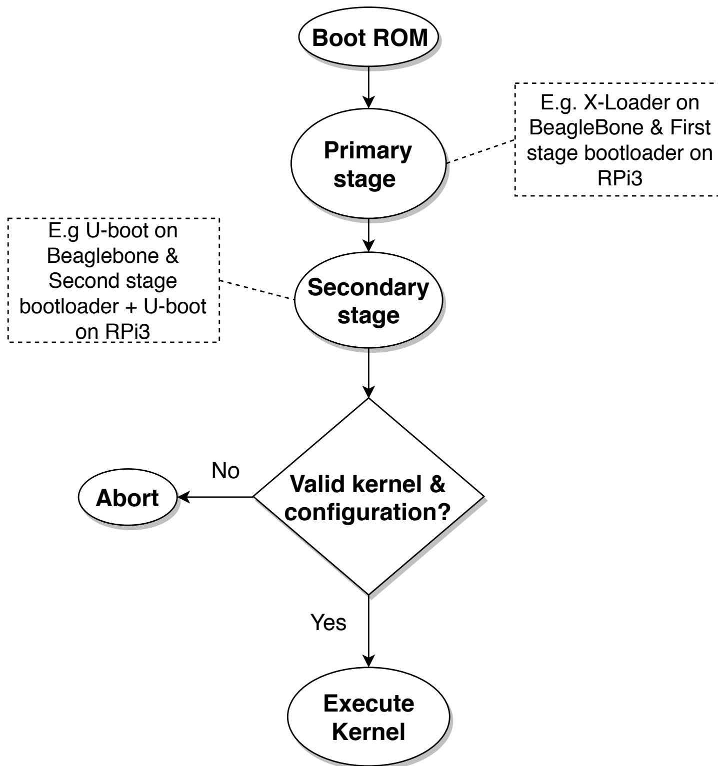


Figure 4.3: Secure boot sequence with U-Boot: U-Boot runs as the second stage, which verifies the kernel and the chosen configuration. U-Boot passes the control of the system to the operating system only after successful verification.

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 [144]. 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 [144]. U-Boot supports three cryptographic hash functions, namely MD5, SHA-1 & SHA-256 [144]. 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 [144] and Elliptic Curve Cryptography (ECC) [144]. 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 more difficult to break. On the other hand, the larger the key size, the longer the verification takes [144].

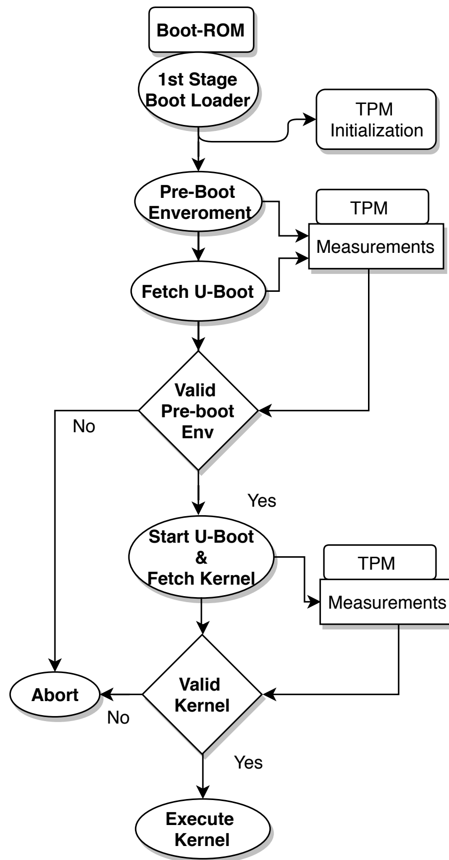


Figure 4.4: Secure boot with a TPM co-processor: The manufacturer provides the boot-ROM and first stage bootloader. We split the second stage into two phases: 1) The pre-boot environment, where we check the integrity of U-Boot. 2) U-Boot execution, where we check the integrity of the operating system.

4.4.2 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. [96]. 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 [145]: (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 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 4.2). The process consists

Table 4.2: 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

of a repetitive **Measure–Extend–Execute** procedure [96]. This method is a common way to ensure a **Chain of Trust** [146], 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 4.4.

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.

4.5 Evaluation

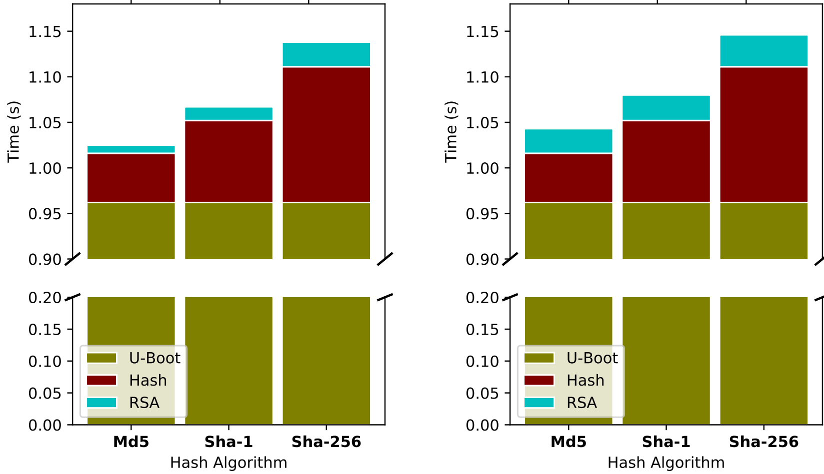
This section evaluates both system designs in detail and focuses on the overhead of the secure boot process in each system. Practical application scenarios motivate this evaluation, for example, vehicles are expected to be immediately usable after ignition, imposing a low-latency requirement between 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.

4.5.1 Experimental Setup

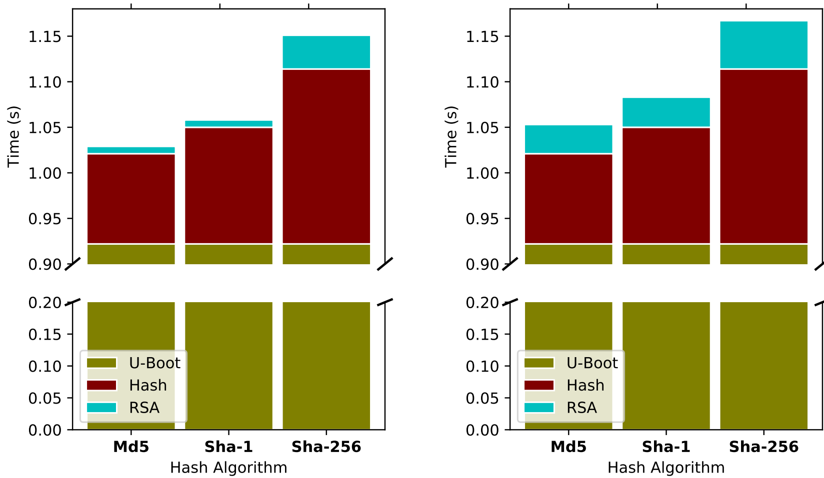
We implement our system design on two generic embedded platforms for IoT applications [147] that are readily available, namely a Raspberry Pi and a BeagleBone (see Table 4.2). Finally, we extend the capabilities of the BeagleBone to support hardware cryptographic primitives.

4.5.2 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

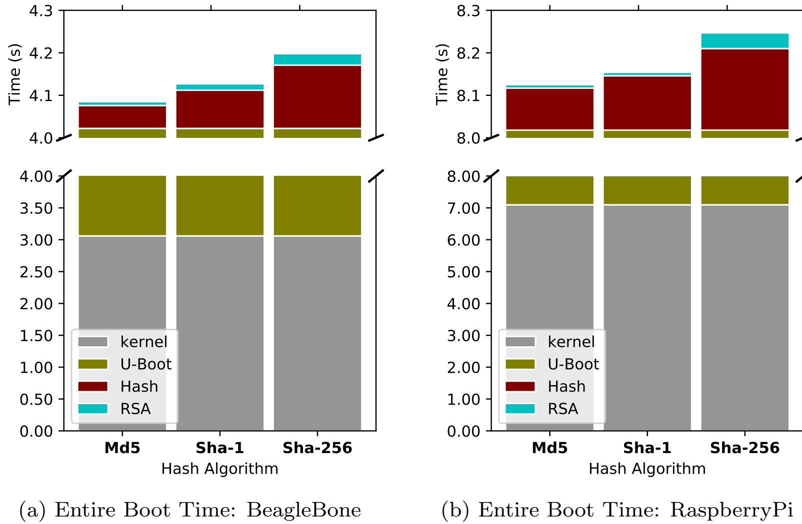


(c) RSA 2048-bits key-size on RaspberryPi (d) RSA 4096-bits key-size on RaspberryPi

Figure 4.5: 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

4.5.2.1 Software Mechanism of U-Boot

Figures 4.5a, 4.5b, 4.5c & 4.5d 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



(a) Entire Boot Time: BeagleBone (b) Entire Boot Time: RaspberryPi

Figure 4.6: 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

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 4.6).

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 4.5a), and 1.4% of the entire boot time of the Linux kernel (see Figure 4.6a). 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 4.5a) and the entire boot time of the Linux kernel (see Figure 4.6a), respectively. For SHA-256 hash function, the overhead increases to 164 ms, 15% and 4%, respectively (see Figure 4.5a & 4.6a).

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 4.5b). For the same experiment using Raspberry Pi, we observe similar results (see Figures 4.5c & 4.5d).

4.5.2.2 TPM Hardware on BeagleBone

Table 4.3 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 4.2) is 138ms. For extending the Registers (PCR) TPM takes 791ms. Overall, the whole method takes 1923ms, which adds an overhead of 36%.

Table 4.3: Verification overhead of TPM

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

4.5.3 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 [148]. FIPS 180-4 Secure Hash Standard (SHS) recommends SHA-1 and SHA-256, but Stevens et al. [149] 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.

4.6 Limitations and Discussion

In this Chapter, 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 Chapter, 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 4.5. 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.

4.7 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 Chapter 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).

4.8 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".

Part III

Embedded Deep Learning on IoT Devices

Paper D

Christos Profentzas, Magnus Almgren, Olaf Landsiedel,

Performance of Deep Neural Networks on Low-Power IoT Devices

Proceedings of the 4th Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench) 2021.

CHAPTER 5

Performance of Deep Neural Networks on Low-Power IoT Devices

Advances in deep learning have revolutionized machine learning by solving complex tasks such as image, speech, and text recognition. However, training and inference of deep neural networks are resource-intensive. Recently, researchers made efforts to bring inference to IoT edge and sensor devices which have become the prime data sources nowadays. However, running deep neural networks on low-power IoT devices is challenging due to their resource-constraints in memory, compute power, and energy. This Chapter presents a benchmark to grasp these trade-offs by evaluating three representative deep learning frameworks: uTensor, TF-Lite-Micro, and CMSIS-NN. Our benchmark reveals significant differences and trade-offs for each framework and its tool-chain: (1) We find that uTensor is the most straightforward framework to use, followed by TF-Micro, and then CMSIS-NN. (2) Our evaluation shows large differences in energy, RAM, and Flash footprints. For example, in terms of energy, CMSIS-NN is the most efficient, followed by TF-Micro and then uTensor, each with a significant gap.

5.1 Introduction

In recent years, Deep Neural Networks (DNNs) have outperformed other algorithms to solve complex problems in computer vision [14], Natural Language Processing (NLP) [150], and Human Activity Recognition (HAR) [151]. Similarly, novel IoT applications utilize DNNs to recognize and categorize sophisticated sensor data [152]. Typically, the resource-intensive tasks of training and inference are offloaded to cloud services. With the Internet of Things, billions of devices produce massive volumes of data to be analyzed, raising two primary concerns. First, IoT devices collecting sensitive sensor data from users and storing them in cloud services raises several privacy issues. Second, processing extensive amounts of sensor data can overwhelm infrastructure in

terms of bandwidth and available computation. To address these concerns, researchers bring inference to edge and sensor devices [153, 154].

In particular, DNN inference on low-power IoT devices brings new challenges due to their resource-constraints. We recognize three main challenges. First, IoT devices have a small memory-size, typically in the range of KBs, where an average DNN requires MBs of storage. Second, DNN inference demands significant energy, but IoT devices require power duty cycling to preserve energy. Third, DNNs are designed using GPU/CPU optimization libraries (e.g., CUDA/Intel DL Boost), unavailable on low-power IoT devices.

The development process of DNNs on low-power IoT devices is a tedious task, where devices are manually managing memory and computation resources. There is a diversity of languages and frameworks, for example, for training DNNs in the cloud or for converting them for IoT devices, and each framework has different tool-chains. Moreover, when designing and training a DNN, it is unknown how efficient it will be on an embedded device. In this Chapter, we argue that it is essential to evaluate and reveal trade-offs of common frameworks, including the complexity of the development process and the resource-efficiency of their IoT run-time environments.

This Chapter focuses on the inference part of Deep Neural Networks (DNNs) on low-power IoT devices, assuming the training is done off-line on more powerful devices. We present a benchmark for evaluating three deep learning frameworks for IoT devices: TensorFlow-lite-Micro [50], uTensor [155], and CMSIS-NN [9]. The Chapter presents the following contributions:

- We design and implement a publicly available[1] benchmark to evaluate the performance of three deep learning frameworks for low-power IoT devices.
- We compare and report each framework's differences, including the development process complexity and the resource-efficiency of their run-time environment on low-power devices.
- We evaluate the resource-efficiency of prevailing DNNs on low-power IoT devices in terms of memory, computation, and energy consumption.

The rest of the Chapter is organized as follows. Sec. 5.2 provides the necessary background. Sec. 5.3 introduces our benchmark. Sec. 5.4 presents our evaluation results. Sec. 5.5 discusses related work and Sec. 5.6 concludes this Chapter.

5.2 Background and Motivation

In this section, we provide the necessary background on deep learning for low-power IoT devices.

5.2.1 Convolutional Neural Networks

In this chapter, we focus on the widely supported Convolutional Neural Networks (CNNs). Other architectures like Recurrent Neural Networks (RNN) [40]

[1] https://github.com/chrpro/TinyML-Evaluation/

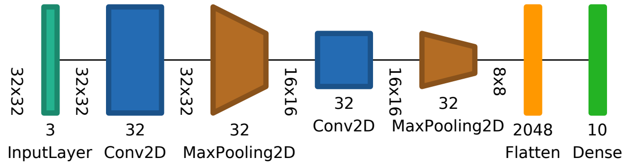


Figure 5.1: Overview of a Convolution Neural Network (CNN). The CNN consists of several layers stacking together: convolution kernels, max pooling, and fully connected layer. The final layer is the output number of classes.

Table 5.1: High-level comparison of deep learning frameworks. The code generator is different across the platforms. The dependency refers to other libraries needed to run inference. The usability reflects the end-to-end development process (from training to on-device inference).

Framework	Code	Generator	Dependency	Usability
uTensor	C++	Auto	Mbed OS	Easy
TF-Micro	C++	Auto	None	Medium
CMSIS-NN	C	Manual	None	Hard

have limited support by the three considered frameworks. Figure 5.1 illustrates the architecture of a CNN. The networks consist of repetitive layers of convolution kernels and pooling operations [40]. A *convolution kernel* is a matrix applying a linear transformation to an image. *Pooling operations* are down-sampling the matrix to lower dimensions by summarizing the essential features. The training process finds the values of the weights of the matrices by minimizing an objective loss function [40].

5.2.2 Quantization

A quantized neural network converts the weights from high precision floating points to integers. The mapping function essentially reduces the number of bits used to represent the weights. The choice of the quantization method is an open research question [156, 157].

5.2.3 IoT Deep Learning Frameworks

The frameworks used in our benchmark are the following.

uTensor. An open-source framework [155] for deep learning inference on Mbed-OS enabled IoT devices. uTensor focuses on rapid-prototyping from TensorFlow-trained neural networks to convert them for IoT devices. uTensor is written in C++ and provides built-in functions for quantization.

TensorFlow Lite Micro (TF-Micro). An open-source framework [50] for supporting machine learning inference on micro-controllers. The library is written in C++ as part of the TensorFlow ecosystem. TF-Micro uses TensorFlow to write and train a neural network. TF-Micro provides a wide range of options to optimize and quantize a neural network using the TensorFlow

Lite Converter. TF-Micro can run as stand-alone or using an operating system like Zephyr OS.

CMSIS-NN. Cortex Microcontroller Software Interface Standard for Neural Networks is an open-source library [9] for ARM devices. It is written in C and provides several quantized functions like Convolution, Pooling, Softmax, and Fully-Connected layers. CMSIS-NN does not provide training tools or generators for the C code. The developer needs to use another library to train, quantize, and convert the network's weights in C code. CMSIS-NN is designed to maximize neural networks' performance on the Cortex-M series by employing the on-board DSP accelerator (CMSIS-DSP) [9].

5.3 Benchmark Design

In this section, we introduce the design goals of our benchmark and then present the actual benchmark.

5.3.1 Overview and Evaluation Goals

Our benchmark evaluates three deep-learning platforms: CMSIS-NN, uTensor, and TF-Micro (see Table 5.1). Their tool-chains are similar in training a network but differ significantly on the code generation and quantization methods. Our goal is to evaluate trade-offs between the development process and their run-time environment's efficiency on low-power IoT devices. For the run-time environment, we focus on the efficiency of running inference on low-power devices in terms of: a) memory footprint, b) inference execution time, and c) energy consumption.

5.3.2 Benchmark Process

We present the process of our benchmark in Figure 5.2. We color the automated steps in green and user-defined steps in blue. There are four steps to generate a neural network and execute it on an IoT device. First, we define and train the neural networks and report the accuracy in Keras[2] (see Table 5.2). Keras is an intuitive API wrapper on top of TensorFlow to help define and train machine learning models. All the networks are based on the same architecture (see Figure 5.1), data-set, and hyper-parameters. Second, we quantize the same network for each tool-chain based on each framework's method. Third, we link the network, the evaluation code, and the framework library to produce the executable file. Fourth, we run inference on the same low-power IoT device for each framework to evaluate its performance.

5.4 Performance Evaluation

This section presents our evaluation results on low-power IoT devices. The evaluation answers the following questions: (a) To what extent is inference technically feasible on IoT devices? (b) What is the overhead of neural networks

[2]https://keras.io/

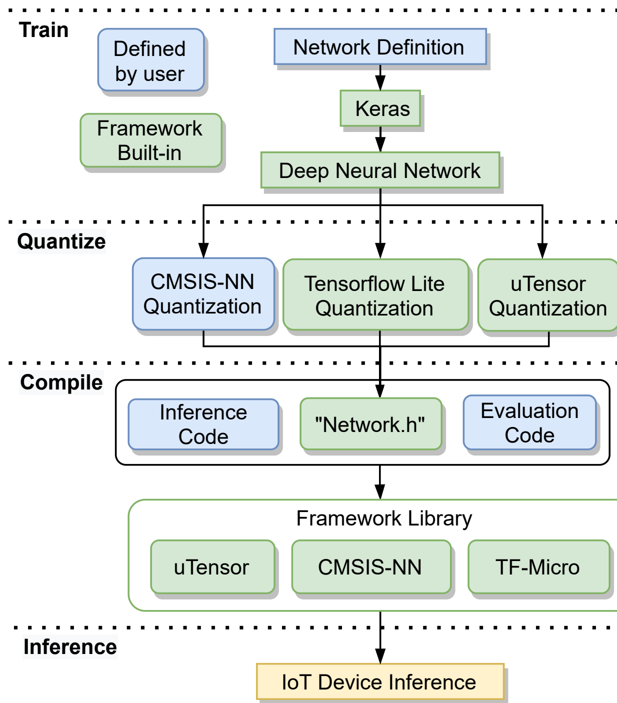


Figure 5.2: Benchmark's development process. The blue boxes are steps defined by the user. The green boxes are automated by the framework. We define and train all networks using Keras. We apply the quantization method provided by each framework. Finally, we compile the code for the target hardware and evaluate the inference for each framework.

Table 5.2: Convolutional Neural Networks trained on MNIST & CIFAR-10. The first column is the number of convolution kernels. For example, Conv-16 means the second layer (see Figure 5.1) and fourth layer have 16 kernels respectively. *Param.* is the number of trainable parameters.

Convolution	**MNIST**		**CIFAR-10**	
- Kernels	Param.	Accuracy	Param.	Accuracy
Conv-16	6,490	0.97	8,538	0.67
Conv-32	17,578	0.98	21,674	0.68
Conv-64	53,578	0.98	61,771	0.70
Conv-96	108,010	0.99	120,298	0.71
Conv-128	180,874	0.99	197,258	0.72

on IoT devices in terms of computation, memory, and energy consumption? (c) What is the trade-off between automation in the development process and performance of inference on IoT devices?

5.4.1 Experimental Setup

Software implementation. We define and train the neural networks listed in Table 5.2 using Python 3.8.6 and Keras 2.4.0. For generating the object code, we have three cases:

- **CMSIS-NN.** We use the ARM GCC toolchain to build and link the C code.

- **uTensor.** We use utensor-cgen to generate the C++ code and mbed-cli to build and link C++ code, including Mbed OS.

- **TF micro.** We use the TF-Lite-Converter to export the network. We use West toolkit from Zephyr OS to build and link the TF-Micro C++ code.

Hardware Setup. We use the nRF-52840-DK board that features a 32-bit ARM Cortex-M4 at 64 MHz supporting DSP instruction set (CMSIS-DSP), 256 KB of RAM, and 1 MB of flash memory, We use the Nordic-Semiconductors Power Profiler Kit v1.1.0[3] to measure power consumption.

Data & Code Availability. We provide the data and code of the benchmark in a public repository.[4]

5.4.2 Data Sets

We train the neural networks with two standard data-sets.

MNIST. The Modified National Institute of Standards and Technology (MNIST) data-set for hand-written numbers. The data-set consists of 70,000 black and white 28x28 pixel hand-written numbers, where 60,000 are for training and 10,000 for testing. There are ten classes, one for each digit.

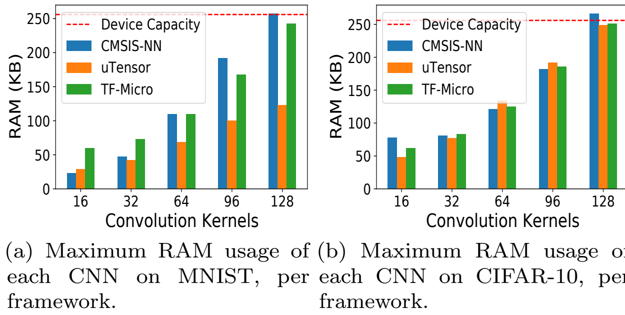
CIFAR-10. The Canadian Institute For Advanced Research (CIFAR) data-set for image classification. The data-set consists of 60,000 32 x 32 color images, where 50,000 are for training and 10,000 for testing. There are ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

5.4.3 Convolutional Neural Networks

For our experiment, we use the networks listed in Table 5.2. Each network consists of seven layers (see Figure 5.1), including the input and output layers. The second layer is a convolution filter with kernel size from 16 to 128, and ReLU activation function. The third layer is a max-pooling operation. The fourth layer is a convolution filter with kernel sizes from 16 to 128, and ReLU activation function. The fourth layer is a max-pooling operation. The names in Table 5.2 reflect the number of kernels. For example, Conv-16 means that the second and fourth layers have 16 kernels, respectively. The motivation is to scale the convolution kernels, which have most of the trainable parameters [40].

[3]https://www.nordicsemi.com/Software-and-tools/Development-Tools/Power-Profiler-Kit

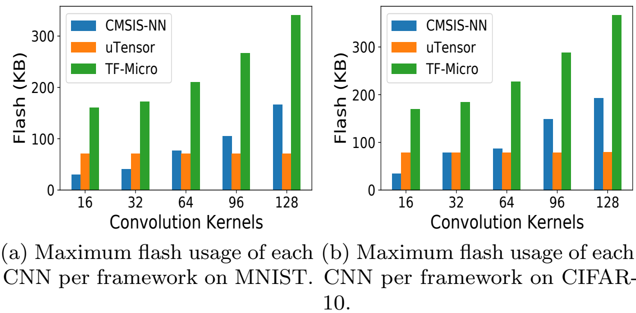
[4]https://github.com/chrpro/TinyML-Evaluation/



(a) Maximum RAM usage of each CNN on MNIST, per framework.

(b) Maximum RAM usage of each CNN on CIFAR-10, per framework.

Figure 5.3: RAM footprints for CMSIS-NN, uTensor, and TF-Micro. Each CNN differs by the number of the convolution kernels (see Table 5.2). The RAM capacity is 256 KB.



(a) Maximum flash usage of each CNN per framework on MNIST.

(b) Maximum flash usage of each CNN per framework on CIFAR-10.

Figure 5.4: Flash footprints for CMSIS-NN, uTensor, and TF-Micro. Each CNN differs by the number of the convolution kernels (see Table 5.2). The flash capacity is 1MB.

5.4.4 Benchmark Evaluation

We repeat each experiment 20 times. We report the maximum values of memory allocation for RAM and flash; we refer to them as footprints. We report the average inference execution time based on the cycle clock register. We report the energy consumption based on the average electric current drawn in mA, by applying 3.3 V, and report the standard deviation as $\pm$, when it is significant.

Memory Footprints. In Figures 5.3, we present the RAM consumption by reporting the maximum memory allocation (footprints), for each network. In Figures 5.3a, we notice that for a small network with 16 kernels trained on MNIST, CMSIS-NN has a small memory footprint (22 KB) similar to uTensor and TF-Micro (28 KB & 60 KB). As we increase the kernels to 128, we notice CMSIS-NN has the largest footprint (253 KB) compared to uTensor and TF-Micro (123 KB & 243 KB). The reason is that CMSIS-NN statically allocates all the necessary data in the RAM. TF-Micro statically allocates memory for the network and dynamically allocates the buffers to store intermittent-results. uTensor dynamically allocates memory both for the network and local buffers (by utilizing Mbed OS). We notice a similar behavior when using the CIFAR-10

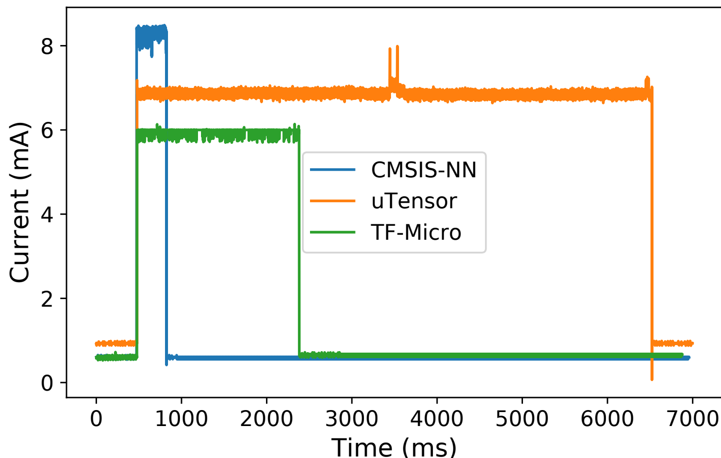


Figure 5.5: A representative sample of electric current drawn during inference on nRF-52840-DK, operating at 3.3 V. The neural network in this example experiment has 16 convolution kernels (see Table 5.2) and is trained on CIFAR-10.

data-set (see Figure 5.3b). However, a network with 128 kernels consumes almost all the available RAM, and with CMSIS-NN (265 KB), it exceeds the device capacity (256KB). However, we consider these networks only for demonstration purposes. A real-world application using these network will not leave any memory for the applications themselves.

In Figure 5.4, we present the flash footprints for each framework. For CMSIS-NN and TF-Micro, the flash footprint increases proportionally with the number of static variables. The reason is that the linker allocates static variables into the flash to be copied to RAM during the start-up phase. For uTensor, the flash footprint is the same for all experiments as it counts only for program code and input vectors. Finally, we notice that the larger neural network (with 128 kernels) consumes less than 37% (367 KB) of available flash (1MB).

Inference Execution Time. We continue with the average inference execution time for each network. In Table 5.3, we report the results for the MNIST data-set. We notice that CMSIS-NN outperforms all other frameworks, as it is specialized for the on-board DSP accelerator available on Cortex-M devices. For example, the smallest network with 16 kernels and CMSIS-NN runs on average in 0.2 s (see Tables 5.2), with TF-Micro in 1.0 s, and with uTensor in 3.0 s. The largest network with 128 kernels and CMSIS-NN runs on average in 8.45 s, with TF-Micro in 43.4 s, and with uTensor in 139 s. In Table 5.4 we report the results for CIFAR-10 data-set. We notice a similar behavior with CMSISS-NN outperforming the others, following by TF-Micro, and uTensor. We have not noticed any significant time deviation among the experiments.

Energy Consumption. In this part, we present the energy consumption based on the average electric current drawn in mA reported by the Power Profiler Kit. Figure 5.5 presents a representative comparison of the electric

Table 5.3: The complete evaluation using MNIST. The RAM and flash footprints are the maximum memory allocation. Time refers to the average inference execution time. The energy consumption is based on the average electric current draw.

	CMSIS-NN				**uTensor**				**TF-Micro**			
Conv. -Ker.	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)
16	22,788	29,724	188	5±0.1	28,544	70,712	3,034	70±0.1	60,432	159,372	973	19±0.1
32	47,396	40,828	597	16±0.1	42,064	70,776	10,001	231±0.2	72,640	171,580	3,218	64±0.1
64	110,436	76,828	2,091	57±0.1	69,104	70,840	34,960	808±0.2	110,880	209,820	11,526	228±0.2
96	191,908	105,336	4,495	122±0.1	100,072	70,904	76,338	1,763±0.2	167,552	266,492	24,939	494±0.2
128	253,042	166,470	8,445	230±0.2	123,184	70,968	138,668	3,203±0.2	242,656	341,596	43,415	860±0.2

Table 5.4: The complete evaluation using CIFAR-10. The RAM and flash footprints are the maximum memory allocation. Time refers to the average inference execution time. The energy consumption is based on the average electric current draw.

	CMSIS-NN				**uTensor**				**TF-Micro**			
Conv. -Ker.	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)
16	77,720	34,076	357	10±0.1	47,704	78,248	5,986	138±0.1	62,048	170,544	1,961	39±0.1
32	80,856	78,360	1,004	27±0.1	76,504	78,264	17,148	396±0.2	82,400	184,800	5,720	113±0.1
64	120,952	87,336	2,100	58±0.1	134,104	78,280	56,824	1,313±0.2	124,736	227,136	18,641	369±0.2
96	182,288	148,660	6,021	164±0.1	191,704	78,344	119,265	2,755±0.2	185,536	287,936	38,775	768±0.2
128	**265,684**	193,068	-	-	249,304	78,472	209,073	4,830±0.2	251,736	367,136	66,113	1,309±0.2

current drawn during inference of a network with 16 kernels trained on CIFAR-10 data-set. We notice a similar trend among all our experiments. CMSIS-NN has the highest current by drawing an average of 8 mA, following by uTensor with an average of 7 mA, and finally TF-Micro with an average of 6 mA. In the same figure, we see that the execution time differs strongly between the frameworks, with CMSIS-NN being the fastest due to DSP acceleration usage.

In Tables 5.3, we report the overall energy consumption for networks trained on MNIST data-set. The energy consumption is strongly related to the inference time and the electric current drawn during that time. We notice that CMSIS-NN has a lower energy consumption among all frameworks. For example, the smaller network with 16 kernels on CMSIS-NN consumes 5±0.1 mJ on average, followed by TF-Micro with 39±0.1 mJ on average, and uTensor with 70±0.1 mJ on average. For a large network with 128 kernels, CMSIS-NN consumes 230±0.2 mJ on average, followed by TF-Micro with 860±0.2 mJ on average, and uTensor with 3,203±0.2 on average. We notice a deviation because of the dis/charging of the PPK capacitors between the experiments. In Table 5.4 we report the results for CIFAR-10 data-set. We notice a similar trend with CMSISS-NN having the lowest energy consumption, following by TF-Micro and uTensor with the highest energy consumption.

5.4.5 Discussion and Limitations

We now discuss our results and remaining challenges.

Framework automation. We observe that defining and training a deep neural network follows a similar process among all frameworks. However, they differ significantly in development process automation and especially in the

quantization method. uTensor offers an easy-to-use tool-chain where built-in functions are handling the quantization and code generation. TF-Micro offers more sophisticated quantization methods, but the user needs to configure them manually. CMSIS-NN gives programmers full-control over every aspect, leading to very efficient code, but the networks need to be converted and defined manually in C code. The result is a trade-off between the complexity of the development process and the efficiency of the resulting code-base: Code in CMSIS-NN has a higher efficiency but is more complex to implement when compared to simpler and versatile approaches (TensorFlow ecosystem) with less focus on performance.

Memory Management. We observe that memory management plays a crucial role in the performance of low-power IoT devices. CMSIS-NN benefits from static allocation of memory regions and definition of specialized 8-bit and 16-bit DSP data-types. In addition, CMSIS-NN boosts performance on the Cortex-M series by employing the on-board DSP accelerator (CMSIS-DSP). On the other hand, uTensor and TF-Micro use generic 32-bit data types and dynamic memory allocation. The generic data type allows multiple platform support, but it comes with a cost on low-power IoT devices.

Concluding Remarks. We conclude the discussion with two remarks. First, there is no framework to fit both the rigorous development process of deep neural networks and the low-power devices' performance. Second, wide networks consume significant memory and energy of devices across all frameworks. Low-power IoT devices can not utilize well-established pre-trained networks available on cloud services. There is a need for ultra-lightweight neural network architectures for low-power devices.

5.5 Related Work

Compression. The frameworks of our evaluation supports only quantization to reduce the size of the networks. Other unsupported methods include weight pruning [156], and network compression [156].

ML Frameworks. In this Chapter, we use three popular frameworks for low-power devices. Next to these, other framework exist: **STM32 Cube.AI [158]** is a proprietary machine learning framework by STMicroelectronics. This framework has a limited scope and is tied to STM MCUs. **Glow [159]** is an open-source machine learning graph optimizer created by Facebook. Glow does not support low-power devices yet, but it is an on-going work.

ML Benchmarks. Our benchmark is not the first one to evaluate the performance of machine learning platforms. **MLPERF [160]** is a large-scale benchmark suite for Machine Learning inference across different platforms and hardware. **DAWNBench [161]** is a deep learning benchmark focusing mostly on GPU performance. In contrast, our Chapter focuses on the applicability of DNN inference on low-power IoT devices.

5.6 Conclusion

This Chapter shows the trade-offs between the development process automation of Deep Neural Networks (DNNs) and the low-power devices' performance. We

present a benchmark to evaluate three representative frameworks for DNNs inference on low-power IoT devices. Our benchmark reveals significant differences and trade-offs for each framework and its tool-chain: (1) We find that uTensor is the easiest framework to use, followed by TF-Micro, and then CMSIS-NN. (2) Our evaluation shows large differences in energy, RAM, Flash footprints. In terms of energy, CMSIS-NN is the most efficient, followed by TF-Micro and then uTensor, each with a significant gap.

Paper F

Christos Profentzas, Magnus Almgren, Olaf Landsiedel

MicroTL: Transfer Learning on Low-Power IoT Devices

Proceedings of the 47th IEEE Conference on Local Computer Networks (LCN), 2022

CHAPTER 6

MicroTL: Transfer Learning on Low-Power IoT Devices

Deep Neural Networks (DNNs) on IoT devices are becoming readily available for classification tasks using sensor data like images and audio. However, DNNs are trained using extensive computational resources such as GPUs on cloud services, and once being quantized and deployed on the IoT device remain unchanged. We argue in this paper, that this approach leads to three disadvantages. First, IoT devices are deployed in real-world scenarios where the initial problem may shift over time (e.g., to new or similar classes), but without re-training, DNNs cannot adapt to such changes. Second, IoT devices need to use energy-preserving communication with limited reliability and network bandwidth, which can delay or restrict the transmission of essential training sensor data to the cloud. Third, collecting and storing training sensor data in the cloud poses privacy concerns. A promising technique to mitigate these concerns is to utilize on-device Transfer Learning (TL). However, bringing TL to resource-constrained devices faces challenges and trade-offs in computational, energy, and memory constraints, which this paper addresses. This paper introduces MicroTL, Transfer Learning (TL) on low-power IoT devices. MicroTL tailors TL to IoT devices without the communication requirement with the cloud. Notably, we found that the MicroTL takes 3x less energy and 2.8x less time than transmitting all data to train an entirely new model in the cloud, showing that it is more efficient to retrain parts of an existing neural network on the IoT device.

6.1 Introduction

Compression methods like quantization and pruning [9, 20, 49] have enabled the deployment of Deep Neural Networks (DNNs) on resource-constrained devices [9–11]. However, the training, optimization, and compression of neural networks are commonly conducted before deployment using vast computational resources like GPUs or even cloud services. Once trained, compressed, and

deployed, DNNs commonly remain static. We argue in this Chapter that this approach leads to three drawbacks. First, IoT devices are deployed and interact with real-world environments, and the initial domain may shift to a different distribution [22]. For example, a user might want to add or remove a class, such as adding a new exercise activity on her smartwatch. With current methods, deployed neural networks on IoT devices [9–11] cannot adapt to such changes. The typical approach is to (re)train a new DNN on the cloud, optimize it for the local hardware (e.g., using post-quantization [9, 10, 20] or quantization-aware learning [21]), and re-download it from edge or cloud services. Due to the large amount of data needed for training [40] and the non-negligible size of the deep neural networks, an IoT device needs to spend significant energy sending training samples and downloading new models from cloud services each time the problem domain changes. Second, resource-constrained devices depend on energy-efficient communication with limited reliability and network bandwidth. In practice, this may delay or even prohibit the transmission of training samples from the device. Third, uploading sensor data to the cloud poses privacy concerns. As cloud services struggle to address privacy issues, they may aggregate data from different sources, where keeping independent and identically distributed (iid) samples along with efficient communication is an open challenge [52].

To avoid communication costs and address problem domain shifts, we can use Transfer Learning (TL) [81]. TL utilizes pre-trained deep neural networks and re-trains parts of the network using a smaller dataset [81] for a smaller but related problem. Today, TL is typically done on uncompressed floating-point DNNs (i.e., non-quantized), with high dimensional training data. Bringing transfer learning to quantized DNNs deployed on resource-constrained devices creates trade-offs in computational, energy, and memory constraints and leads to two main challenges. First, resource-constrained IoT devices have minimal memory, such as 64-256 KBs of RAM and 32-64 MHz CPU, often operate on batteries, and are unable to store and process high dimensional data. Second, typical quantize-aware training methods [21] retain the precision of gradients by duplicating all values of the neural networks, one with floating points and one with quantized values, respectively. This double-booking prohibitively increases the memory and computational for resource-constrained IoT devices.

We present MicroTL, which tackles the above challenges by tailoring Transfer Learning (TL) to resource-constrained IoT devices. MicroTL combines parts of the output of quantized hidden layer(s) of an existing deep neural network with new fully connected layer(s) specialized to the new classes of the problem domain. MicroTL enables (re)training parts of DNNs on the IoT device, and thus no communication with the cloud is required, thereby protecting sensitive information. Moreover, this enables the personalizing of deep neural networks to the end-user using local data while preserving the user's privacy at the same time.

In summary, this Chapter makes the following contributions:

- We enable transfer learning on resource-constrained devices, allowing neural networks to adapt to changes in the problem domain and removing the need to communicate with the cloud.
- We show that combining the outputs of the hidden layer(s) with fully

connected layer(s) is sufficient for learning without duplicating the values of the network.

- We design and implement MicroTL, an open-source[1] transfer learning system for resource-constrained IoT devices. We provide a discussion of implementation challenges and trade-offs.
- We quantify the performance of MicroTL in terms of accuracy, computation, memory, and energy consumption. Notably, we find that for a particular dataset it takes 3x less energy and 2.8x less time than transmitting all the local data to the cloud to create a new model.

Chapter outline. We organize this Chapter as follows. Section 6.2 provides the necessary background. Section 6.3 introduces MicroTL system design. Section 6.4 presents the evaluation of MicroTL. Section 6.5 discusses related work, and Section 6.6 concludes this Chapter.

6.2 Background

In this section, we provide the necessary background on transfer learning and quantization.

6.2.1 Transfer Learning

Transfer learning (TL) aims to reduce the amount of training data and speed up the training process. The motivation of TL [81] is to utilize existing neural networks pre-trained on large data sets for a generic problem domain referred to as the **source domain** and adapt for similar smaller domain referred to as the **target domain**. TL uses the hidden layers of a pre-trained network as feature extractors from the source domain to append and train end-layer(s) on the target domain. A typical approach of transfer learning consists of three steps. First, it removes the Fully Connected (FC) layers at the end of a pre-trained network. Second, it appends a new FC layer(s) with output matching the number of the target domain classes. Third, it freezes the weights (no backward calculations) of all layers prior to FC layer(s) and trains the network using the target domain data set by updating only the weights of FC layer(s).

6.2.2 Quantization

In order to deploy deep neural networks on IoT devices for inference, we need to train and optimize them using another library (e.g., PyTorch [162], Tensorflow [163]) on cloud services. Due to the resource constraints of IoT devices, it is common to compress the neural networks using quantization [21, 164]. Fixed-point quantization represents real numbers with integers using fixed-length fractional parts. Int-8 quantization [9, 10] is the most common approach to compress DNN using 7-bits for fixed-point representation of original values, plus 1-bit for the sign (8-bits in total). It reduces the size without changing the original architecture of the network and affecting the overall

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

accuracy by a margin [9]. The quantized function requires both the scaling of the min-max range of real numbers to integers and a shift offset because the zero-point in the integer domain is different from the real number domain due to the quantization. There are two widely used methods to produce the final quantized version of the network: post-training and quantization-aware training.

Post-training quantization. In this method [9, 10, 20], the neural network is training as usual with floating-point. The network is statically quantized using the min-max range of the weights from floating-point to integers. However, this method needs a representative data-set from the device to calibrate the activation output of each layer. This happens only once, and the weights never change after deployment.

Quantization-Aware Training (QAT). This method [21] simulates the quantization effect during training by duplicating all the weights and bias, one in 8-bit integers for the forward pass, and one in 32-bit floating-points for the backward pass, together with meta-data regarding rounding effects. During the forward-pass, it utilizes the 8-bit integer weights of the network, but during the back-propagation, it calculates the gradient using the 32-bit floating-point weights. At the end of each iteration, it quantizes the floating-point weights to 8-bit integer weights. With this method, the quantization error is part of the learning process providing better accuracy, but the quantization parameters become part of the overall training.

6.3 MicroTL Design

MicroTL's main objective is to personalize deep neural networks to end-user preferences without the communication dependency on sending sensor data from IoT devices to the cloud and neural network updates from cloud to IoT devices. MicroTL allows training fully connected layers personalized to the end-user, and it keeps the data on the device, avoiding sending sensitive or private data to the cloud. MicroTL assumes a pre-initialization step of downloading or flashing a pre-trained neural network on IoT devices, as they are widely available today with CMSIS-NN [9] or Tensorflow-micro [50].

6.3.1 System Challenges

We identify two main system challenges when designing MicroTL. First, deep neural networks for low-power IoT are typically quantization-aware trained or post-quantized to address memory and computational constraints. Even though inference after quantization is straightforward, reversing the effects of quantization for additional learning is an open challenge [165]. Second, low-power IoT devices have memory and computational limitations for storing training samples and computing learning parameters.

6.3.1.1 Learning After Quantization

Due to common gradient descent optimization methods in deep learning [40], any training algorithm needs to calculate and retain the precision of the gradients. Low-power IoT devices lack the resources to keep duplicate weights

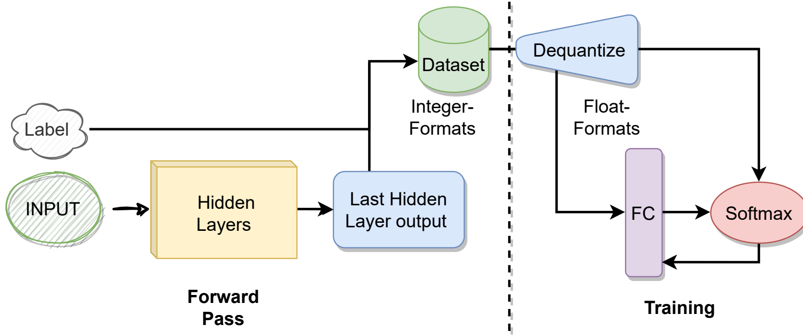


Figure 6.1: The two-step MicroTL approach for transfer learning on IoT devices. First, it stores the output of the last hidden layer in int-8 format during the forward pass. Second, it de-quantizes the samples during training and trains a fully connected layer in floating-point format.

(like QAT, as explained in section 6.2.2), and there are no standard methods to reverse quantized layers for additional learning [165]. The IoT device can only utilize the local quantization scale factors and does not have access to other parameters used during training as opposed to QAT.

6.3.1.2 Device Resource Constraints

The collection of large sample data is restricted on IoT devices primarily due to computational and memory constraints. The forward pass latency on low-power devices is still one of the main bottlenecks for processing samples [11]. Current approaches for transfer learning send the data to the edge and cloud that do not face such resource limitations. These edge or cloud-based methods typically use data sets in high-dimensional inputs, focusing mainly on training latency. Transfer learning on low-power devices needs to address trade-offs between computational, energy, and memory constraints on low-power IoT devices. For example, in the Imagenet data set, the image size is 256x256, occupying 196 KB, almost all the RAM space of a low-power IoT device. Even with smaller image sizes like CIFAR-10, an image (32x32) will occupy 4 KB, which is a significant amount for a low-power device.

6.3.2 MicroTL Overview.

The IoT device has a pre-installed quantized neural network trained for a generic problem (source domain). The IoT device interacts with the environment and collects new sensor data. The IoT application forwards the data to the neural network for inference. MicroTL provides two main operations on top of the quantized network. First, the sample collection takes control of the hidden layer outputs and creates a local training data set. For examples, when a user using a smartwatch and engages in activities, the device creates a training set by using another sensor or asking the user through user interaction. Second, when the training set is filled with enough data, it initiates the training of appended Fully Connected (FC) layer(s). In Figure 6.1, we illustrate the overview of MicroTL.

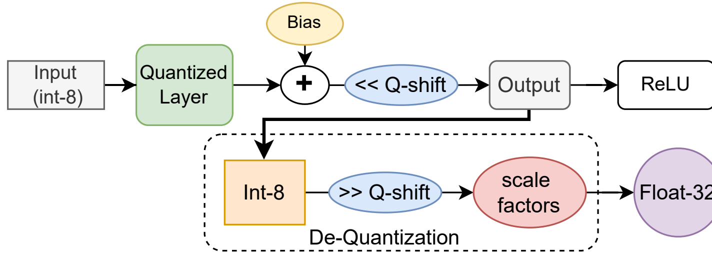


Figure 6.2: Overview of the forward pass on a quantized network together with MicroTL's dequantization operation. The figure demonstrate the operation over a single input. In MicroTL we collect several outputs and perform the de-quantization during training. Q-shift is the offset because of the zero-point difference in the real and integer domains. Scale factors come from the min-max quantization.

6.3.3 Dequantization.

In order to employ training on resource-constrained devices, we enforce the optimization and the proper placement of the dequantization process. In Figure 6.2, we present in detail where we collect the intermittent outputs and which operations are involved. After the convolution operation and adding bias, the matrix is shifted due to zero-point. We collect the output before applying ReLU to avoid consecutive zeros. Due to quantization, we have Q-shift, an offset of the difference between the integer domain's zero-point and real numbers. We need to reverse both the min-max scale factors between integer and real numbers and shift back and reverse the zero-point offset.

6.3.4 Pre-trained Neural Networks

MicroTL utilizes the hidden layers of the existing pre-trained neural network as feature extractors for applying transfer learning of fully connected layers. MicroTL freezes the pre-trained hidden layers and trains only the Fully-Connected (FC) layer(s) depending on the target domain, specialized to the end-user. MicroTL manages the memory allocation together with the computations of forwarding pass for FC layers. The weights of the FC layer are stored in floating-point precision. The floating-point precision enables the computation of adequate gradients for training with the back-propagation algorithm.

6.3.5 Sample Storage

Typical deep learning approaches store high-dimensional training samples on large databases. For low-power IoT devices, this will overwhelm the limited resources. One key observation is that deep hidden layers can reduce the dimensions of the input data. In Table 6.1, we can observe that hidden layer outputs have a smaller dimension (shape) than the input data. Since we train only the weights of the last fully connected layers, in principle, we can pre-compute the outputs of the last hidden layer and store them in a lower dimension data set. MicroTL collects, during inference, the output of the last

Algorithm 2: MicroTL Algorithm

```
    Batch Size: 4 / 8 / 16 / 32
    Pre-install: 8-bit Neural Net
    Constants: Learning rate, Momentum, Decay
 1  // Collecting Training Data
 2  while sensor-data d do
 3      Label := Receive user label y_t;
 4      Forward pass (Net, d);
 5      Hidden layer output := Net.layer(s).output;
 6      // 8-bit int format
 7      TrainingData (y_i, x_i) := (Label, Hidden layer output);
 8  end
 9  // Training Process
10  FC := Randomized Fully Connected Layer(s)
11  for epoch e = 1, ... , E do
12      for batch m in TrainingData do
13          // 32-bit floating point format
14          Y^(i), X^(i) := de-quantize (m, Q-factors, Q-shift);
15          Y^(j) := Forward (FC, X^(i));
16          delta := - sum_{i=1}^{m} y_i log(y_j) // cross entropy loss
17          Back propagation (FC, delta);
18          Update weights (FC, Learning rate, Momentum, Decay);
19      end
20  end
```

convolution or recurrent layer and stores it into 8-bit integer-format data set. The training data set is stored locally and MicroTL does not transmit any sensor data or intermittent results to the cloud. The training data is discarded after training, preserving the privacy of the user.

6.3.6 MicroTL Training Algorithm

MicroTL's training algorithm uses back-propagation with mini-batch stochastic gradient descent and cross-entropy as a loss function to train the fully connected layers. MicroTL uses 32-bit floating-point representation for the fully connected weights to achieve convergence when calculating the gradients in the backward phase. The pseudo-code version of the training process is presented in Algorithm 1. The algorithm works with batches of 8, 16, or 32, depending on the device's available memory. It de-quantizes a batch sample by dividing the Q-factors and bit-shifting with Q-shift (see Figure 6.2). However, a key observation is that the division is a power of 2 which can also be done using bit-shifting, and most cross-compilers can highly optimize and combine these operations. The scale factors and Q-shift are pre-calculated using min-max quantization [49]:

$$q_x = \frac{(2^n - 1)/2}{\max(\text{abs}(X.min), \text{abs}(X.max))}$$

MicroTL temporarily stores the batch for each training epoch. The

learning rate, momentum, and decay are hyperparameters controlling the learning process. The training process randomly initializes the Fully Connected (FC) layer(s) weights. Next, for each epoch, MicroTL de-quantizes the batch, computes the output loss, and performs backpropagation. Finally, the training process minimizes the loss function (cross-entropy) given the hyperparameters.

6.4 Performance Evaluation

This section presents the evaluation of MicroTL on low-power IoT devices. It gives answers to the contributions listed in the introduction on three specific questions: (a) Is transfer learning feasible on low-power IoT devices? (b) What is the accuracy of on-device training compared to edge or cloud-based training? (c) What is the overhead of transfer learning in terms of computation, memory, and energy consumption?

Outline. The evaluation is divided into five parts. First, we present the implementation details. Second, we present the training set and neural network of experiments. Third, we present a comparison of MicroTL with edge and cloud-based approaches. Fourth, we present the evaluation of MicroTL on low-power devices. Fifth, we discuss our results and remaining challenges.

6.4.1 Experimental Setup

Software & hardware implementation. We define, train, and quantize each deep neural network using PyTorch in Python 3.8. The quantized pre-trained layers are listed in Table 6.1. We utilize CMSIS-NN [9] library for the inference part on IoT devices. We implement MicroTL in C, using the Arm-gcc 9.2.1. We evaluate MicroTL on nRF-52840-DK board featuring: a 32-bit ARM Cortex-M4 with an FPU at 64 MHz, a DSP co-processor, 256 KB of RAM, and 1MB KB of flash. The board has wireless communication capability with Bluetooth Low Energy (BLE), Thread, and Zigbee. The chip of this board is widely used on low-power IoT applications like wearable and smart-watches [166]. Finally, we use the Nordic-Semiconductors Power Profiler Kit (PPK) v1.1 [167] to measure the power consumption.

6.4.2 Training Sets and Neural Networks

We use two representative machine learning datasets for image classification and human activity recognition. For image classification, we use CIFAR-10 [168], reflecting computer vision IoT applications. For human activity recognition, we use HAR-UCI [13] reflecting typical IoT health tracker applications.

CIFAR-10. The data-set consists of 60,000 colored 32x32 images, where 50,000 are for training and 10,000 for testing. There are ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. We use seven classes (plane, car, cat, deer, dog, horse, ship), named it CIFAR-7, and use it as source domain. CIFAR-7 has 40,000 training and 7,000 testing images. We use the remaining three mutually exclusive classes (bird, frog, truck) for transfer learning as the target domain. We named the data set CIFAR-3 with 10,000 training and 3,000 test images.

Table 6.1: The neural networks used in our experiments. The table shows the shape size, and the layer output in bytes for each layer.

Pre-trained RNN on HAR-UCI-3a		
Freezed Layers	Shape	Bytes
Input	(128, 9)	1,152
Conv1D	(43, 9)	387
LSTM	(43, 32)	1,376
GRU	(43, 64)	5,504
GRU	(43, 32)	2,752
GRU	(43, 6)	258
Trainable Layers (MicroTL)		
Fully Connected	(258, 32)	33,024
Fully Connected	(32, 3)	384

Pre-trained CNN on CIFAR-7		
Freezed Layers	Shape	Bytes
Input	(32, 32, 3)	3,072
Conv2D	(30, 30, 64)	57,600
MaxPool	(15, 15, 64)	14,400
Conv2D	(13, 13, 32)	5,408
MaxPool	(6, 6, 32)	1,152
Conv2D	(4, 4, 16)	256
Trainable Layers (MicroTL)		
Fully Connected	(256 , 32)	32,768
Fully Connected	(32, 3)	384

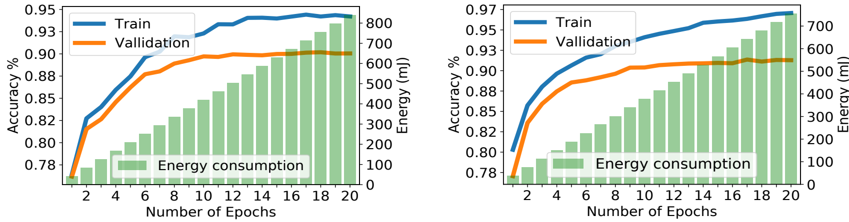
Table 6.2: Average accuracy over the complete test set reported as a percentage (%). Uplink refers to the data we need to be transmited in KB.

Training Sample	**HAR-UCI-3b** Accuracy (%) MicroTL	Edge-TL	Uplink (KB)	**CIFAR-3** Accuracy (%) MicroTL	Edge-TL	Uplink (KB)
100	78±0.4	77±0.4	115	84±0.8	83±0.3	307
200	82±0.2	80±0.2	230	88±0.3	84±0.2	614
300	83±0.1	81±0.1	346	89±0.2	84±0.2	922
400	85±0.1	82±0.1	461	91±0.2	87±0.1	1,230
500	87±0.1	83±0.1	576	92±0.1	88±0.1	1,540
600	87±0.1	83±0.1	692	92±0.1	88±0.1	1,840
ALL (Cloud-QAT)	88±0.1		4,210	93±0.1		46,100

HAR-UCI. The data-set is a human activity recognition data set. HAR-UCI is based on time-series of sensors (accelerometer and gyroscope) captured by smartphone (Samsung Galaxy S II), and it has six classes (activities): 1) walking, 2) walking upstairs, 3) walking downstairs, 4) sitting, 5) standing, 6) laying. The data type is a time series of signals needing pre-processing before use. Our experiments parse the data per 128 time-step and apply min-max normalization over the nine-axis (final dimension 128x9), leading to 7,300 training samples and 3,000 testing samples. We use the following three classes as the source domain (*HAR-UCI-3a*): walking upstairs, walking downstairs, and standing. HAR-UCI-3a has 3,650 samples for training and 1,500 for testing. We use the other three classes as the target domain: walking, sitting, and lying. We named the data set as *HAR-UCI-3b*, and it has 3,650 samples for training and 1,500 for testing.

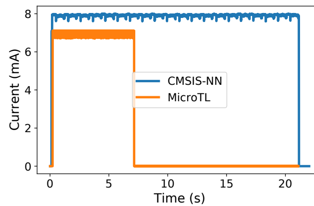
Pre-trained networks. Our experiments use two pre-trained and quantized neural networks based on CMSIS-NN example models [9]. For the CIFAR image classification problem, we utilize a convolutional neural network (CNN). For the HAR-UCI we utilize a Recurrent Neural Network (RNN). We remove the last fully connected layers and append the Fully Connected (FC) layers of MicroTL. We report the shape size and the output of each hidden layer in bytes. Finally, we list the appended Fully Connected (FC) layers for transfer learning (MicroTL).

MicroTL training sets. We draw balanced (all classes equally represented) random samples of size from 100 to 600 from each training set (CIFAR-3,



(a) MicroTL train & validation accuracy together with energy consumption on HAR-UCI-3b.

(b) MicroTL train & validation accuracy together with energy consumption on CIFAR-3.



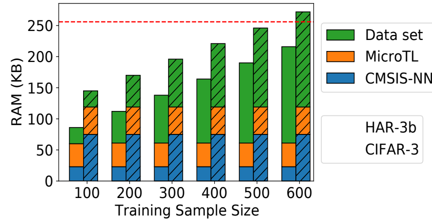
(c) Electric current drawn during forward-pass and training on nRF-52840-DK, by applying at 3 V.

Figure 6.3: Train and validation accuracy on MicroTL using 500 samples and 20 epochs. The green bar refers to the total energy consumption aggregated with each epoch. In the third figure, CMSIS-NN refers to forward-pass and MicroTL refers to training with 100 samples on CIFAR-3.Electric current.

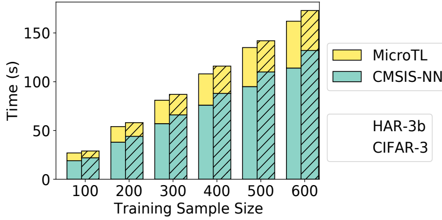
and HAR-UCI-3b). We split the training set to 90% for training and 10% for validation, and we repeat 50 times the training experiments of MicroTL for each training sample size. We use validation-based early stopping [40] to avoid over-fitting. We use momentum [40] to accelerate training and weight decay [40] to adapt learning rates and set the batch size to 8 for all experiments. Finally, we report the standard deviation as ±.

6.4.3 MicroTL Learning Accuracy

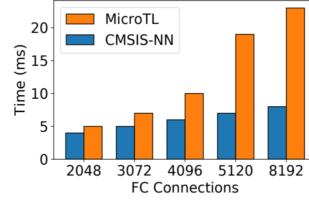
Baselines. Our experiments compare the accuracy obtained with MicroTL on low-power IoT devices (on-device learning) with the accuracy we would have obtained if we have trained in edge/cloud services. We create two baselines. **Edge-TL** is the straightforward approach to send training data and apply Transfer Learning (TL) in the edge, where a copy of the deployed neural network (before quantization) is stored. The transfer learning in the Edge-TL happens in Python libraries (e.g, TensorFlow), and the network is post-quantized to fit the memory constraints of the IoT device. **Cloud-QAT** is an approach where the device sends all training data to the cloud by applying Quantization-Aware Training (QAT) to optimize a new neural network from scratch. For both Edge-TL and Cloud-QAT, we use the same Python version, and we report the results of the network in the final form to be deployed in the IoT device.



(a) The RAM consumption consists of different parts: CMSIS-NN network, MicroTL, and training data set. The red line is the RAM capacity of the device (256KB).



(b) Training time of MicroTL compared to the forward pass. CMSIS-NN refers to forward part of hidden layers, MicroTL refers to training time of the fully connected layers.



(c) Comparison of the inference time with a floating-point fully connected layer (MicroTL) and CMSIS-NN.

Figure 6.4: Performance evaluation of MicroTL in terms of memory, computation and energy consumption.

Communication. For the data transmission of the IoT device to the edge and cloud, we assume a Low-Power Bluetooth (BLE) connection, typically available on low-power devices. We report the transmission latency using the standard throughput of 700 kbps (payload) using the default 1 Mbps mode [166]. For energy, we report the energy consumption using the average current drawn from the BLE radio evaluated in previous work [166] (using PPKv1.1) by applying 3 V, which is 7.5 mA.

Accuracy. In Table 6.2, we present the average test accuracy of MicroTL and baselines, Edge-TL and Cloud-QAT. The accuracy is over the complete test set reported as a percentage (%). With 100 training samples, MicroTL has marginally higher accuracy (on test-set) than Edge-TL, but Cloud-QAT can achieve almost 10% higher accuracy than MicroTL. The accuracy improves with more samples, and MicroTL can reach accuracy close to Cloud-QAT on CIFAR-3 and on HAR-UCI-3b. For example, with 500 training samples, the accuracy on test set of MicroTL on CIFAR-3 is 92% close to 93% of Cloud-QAT. Similarly with HAR-UCI-3b MicroTL achieves 87% close to 88% of Cloud-QAT. Finally, we notice that transfer learning in the Edge (Edge-TL) achieves lower accuracy than MicroTL. The main reason is that the MicroTL carefully collects and de-quantizes intermittent outputs in high precision for the fully connected layers. Edge-TL uses standard post-quantization to produce the final network. On the other hand, Cloud-QAT uses quantization-aware training to simulate and duplicate the quantization effect over the full network and outperforms

both Edge-TL and MicroTL with respect to accuracy, but the communication cost is higher.

MicroTL training cost. We plot in Figure 6.3 the average train and validation accuracy together with energy consumption for 20 epochs, using a 500 training samples, where we achieve high accuracy on MicroTL. The green bar is the total energy consumption aggregated with each training epoch. Figure 6.3a shows the same experiment for HAR-UCI-3b. We achieve 89% validation accuracy after 10 epochs and consume 430 mJ energy. In order to increase the accuracy by a 1-2% margin, we need to spend another 400 mJ. Figure 6.3b shows the same experiment for CIFAR-3. We achieve 88% validation accuracy after 10 epochs and consume 400 mJ. In order to achieve a 1% margin, we need to consume another 335 mJ. However, the training and validation accuracy diverges after 10 epochs, indicating that the network starts over-fitting, and in practice, the algorithm applies early stopping.

Communication cost. In Table 6.2, we report uplink as the amount of data in KB we need to transmit to the edge and cloud (Edge-TL and Cloud-QAT baselines). Downlink is fixed in both baseline, the device needs to receive the new fully connected layers in Edge-TL, which 9 KB for CIFAR-3, 8KB for HAR-UCI-3-b. In the case of Cloud-QAT, the device needs to download a new model, which is 85 KB for CIFAR-3, and 61 KB for HAR-UCI-3-b. However, the device needs to send to the Cloud-QAT a total of 4,210 KB for HAR-UCI-3b, and 46,100 KB data for CIFAR-3. Using BLE for communication will take 50 seconds and consume 1,238 mJ for HAR-UCI-3b, and 552 seconds and consume 13,662 mJ for CIFAR-3. Compared to MicroTL with 500 training samples, achieving similar accuracy, the device will consume a total of 840 mJ and takes 40 s for HAR-UCI-3b. When training on CIFAR-3 it takes 32 s and consumes 735 mJ, on average. Overall, the training process with CIFAR-3, it takes 2.8x less time, and 3x less energy with MicroTL compared to Cloud-QAT to achieve similar accuracy, similarly with HAR-UCI-3b.

6.4.4 MicroTL Performance on Low-Power IoT Devices

This section presents the evaluation of MicroTL in terms of memory, time, and energy consumption. We report the average peak (maximum amount) for memory consumption on RAM and flash. We report the average training time and forward-pass based on the cycle clock register. We report the average energy consumption based on the average electric current draw in mA, by applying 3 V on Power Profile Kit. We repeated each experiment 50 times, and we report the standard variation as $\pm$.

Memory consumption. In Table 6.3, we report the RAM and Flash consumption for each training set. We observe that Flash remains the same through all experiments as it stores only static parts and the code sections. In Figure 6.4a, we demonstrate the average peak memory consumption of RAM divided into three parts: the pre-trained network (CMSIS-NN), transfer learning (MicroTL), and the training data set. The different parts consume memory as follows. CMSIS-NN allocates static memory for storing the quantized network. MicroTL dynamically allocates memory for storing the weights of the fully connected layers and allocates temporary memory for the de-quantized batches during training, along with the gradient matrixes for the backpropagation.

Finally, the data set is the inference output collected during the forward pass of CMSIS-NN in an 8-bit integer format.

For example, with CIFAR-3 data set, the pre-trained network (CMSIS-NN) consumes 30% of RAM, and MicroTL consumes 17% of RAM. The data set size is related to the size of the last hidden output, where on CIFAR-3 is 256 bytes (see Table 6.1). A sample size of 100 requires storing 25,600 bytes of data (10% of the RAM), and a training sample of size 500 consumes 49% of the RAM, while a sample size of 600 together with CMSIS-NN and MicroTL exceeds the memory capacity (red dotted-line). With HAR-UCI-3b, we need 258 bytes per sample (see Table 6.1), a data set of 100 consumes 11%, and a data set of 600 consumes 50% of RAM.

Execution time. Next, we evaluate the execution time of training (MicroTL) and the forward-pass of the pre-trained networks (CMSIS-NN). The training time highly depends on the input of fully connected layers. In all experiments, we use two fully connected layers. The first layer has the input size of the last hidden layer of the pre-trained network and an output of 32 nodes. The final layer takes the 32 nodes and creates an output of the number of classes (specific to each task). In Figure 6.4b, we observe that the average execution of the forward-pass on CMSIS-NN can take up to 25-27% more time than the training part of MicroTL on average. For example, with 100 training samples of CIFAR-3, the forward-pass takes 22 s while the training part of MicroTL takes 7 s on average.

Finally, in Figure 6.4c, we plot the inference time for a quantized fully connected layer (CMSIS-NN) compared to a floating-point (MicroTL). We can see a cost using MicroTL floating-point over fix-point CMSIS-NN. For example, with a shape of 256x32 and 8,192 neuron connections, it takes 23 ms on MicroTL compared to 5 ms on quantized CMSIS-NN. However, the latency is relatively small on the overall inference time, but as we increase the number of neurons, we expect the FC layers to affect the inference time. A complete evaluation of CMSIS-NN is out of scope, and it has been presented by others' work [9, 11].

Energy consumption. The average current drawn during training using MicroTL is lower than the forward-pass of CMSIS-NN, as illustrated in Figure 6.3c. MicroTL draws 7 mA on average using the FPU unit to execute the training process. CMSIS-NN draws 8 mA on average using the DSP co-processor for the inference part. The energy consumption depends on the training time, which takes longer as we increase the training data. In Table 6.3, we report the energy consumption of MicroTL for all experiments. For example, with 600 samples, training consumption can go up to 1,008 mJ on average. However, the accuracy is not improving as we increase the training data, and we get similar accuracy with 500 training samples by spending less energy.

6.4.5 Discussion and Limitations

In this section, we discuss the trade-offs of transfer learning on low-power devices and the remaining challenges.

Hidden layers. The first trade-off we face is the depth of the hidden layers. The pre-trained network needs to have adequate depth so as the hidden layers reduce the input dimension size. However, a deeper network takes more

Table 6.3: The complete evaluation of MicroTL.

	HAR-UCI-3b				CIFAR-3			
Sample Size	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)
100	63	123	8±0.2	168±0.2	70	98	7±0.2	147±0.2
200	89	123	16±0.1	336±0.1	95	98	14±0.1	294±0.1
300	115	123	24±0.1	504±0.1	121	98	21±0.1	441±0.1
400	141	123	32±0.1	672±0.1	146	98	28±0.1	588±0.1
500	167	123	40±0.1	840±0.1	171	98	32±0.1	735±0.1
600	193	123	48±0.1	1,008±0.1	197	98	41±0.1	882±0.1

time to execute on forward-pass. On the other hand, a smaller and faster neural network will need more memory to store the higher dimension output of the hidden layers.

Resource constraints. One of the main challenges of MicroTL is the limited resources in terms of energy and memory storage. We have observed that the training set can occupy up to 50% of RAM, and training can consume up to 1,010 mJ. However, to achieve similar accuracy with other methods, we will need to send sensor data to the edge or cloud. Besides the privacy concerns, the large number of data needed to train a deep neural network will shortly overwhelm low-power IoT devices. For example, sending all training data of CIFAR-3 will require 46 MB. It will take more time and energy to send the data using BLE communication than train on the device using MicroTL.

Number of classes. Our approach works for a relatively small number of classes. We can increase the number of classes by using larger Fully Connected (FC) layers, but it will lead to two issues. First, FC layers will need more data and time to train, and second, it will increase inference time. Overall, solving more complex problems with transfer learning will lead to higher energy consumption.

6.5 Related Work

This section lists the related work regarding different methods for learning on edge devices, and other compression methods for neural networks.

Edge learning. Machine Learning (ML) is primarily associated with cloud services, but recently we have experienced a move from cloud-centric ML, where training and inference happen in the cloud, to learning on edge devices. For example, TensorFlow Lite [163] offers solutions to use on-device learning for mobile phones. Different promising on-device methods have been proposed like transfer learning [169], incremental learning [170], meta-learning [171], and few-shot learning [172]. In all these methods, the device adds new classes or makes predictions based on a limited number of samples. This Chapter focuses on bringing transfer learning from edge to resource-constrained devices.

Transfer learning. DeepCham [83] and IoTTL [169] have shown the feasibility of transfer learning on edge devices (e.g., Raspberry Pi). However, they overlook the challenges of low-power devices, especially the energy and memory limitations, and they do not consider quantized pre-trained networks.

Moreover. their models require MBs of RAM and floating-point precision networks. In contrast, MicroTL tailors transfer learning on low-power IoT devices running on 32-64 MHz and KBs of RAM by addressing their recourse constraints, without needing extra parameters for training and by dynamically utilizing personalized data.

Federated learning. With federated learning, multiple devices train a global neural network using multiple datasets located locally on the device. Training methods of federated algorithms often use Homomorphic Encryption (HE) [86] or Secure Multiparty Computation (SMC) [87] to protect the privacy of the users. Due to this, federated learning today is commonly done on edge class devices. These requirements are demanding for low-power devices. With our work, we bring on-device learning to low-power IoT class devices. Federated learning can be used to create a joint global model, and MicroTL can personalize the model locally on the device.

Network search and compression. Other methods for compression neural network for IoT devices include: pruning [173], Network Architecture Search (NAS) [77] This Chapter focuses on 8-bit quantized methods due to the wide support and success on low-power devices and keeping the accuracy close to the original networks. Even though pruning and NAS can significantly reduce the network's size, they suffer from accuracy drops. Other methods using quantized neural network is feasible, for example, with half-wave Gaussian quantization [78] and low-bit neural networks [48, 79]. However, they have a significant accuracy drop.

6.6 Conclusion

Inference with Deep Neural Networks (DNNs) is readily available on embedded devices. However, the current approaches assume that training and compression of DNNs happen in the cloud. These approaches overlook three key issues. First, IoT devices are deployed and interact in real-world environments, and their initial problem can change. Second, uploading personalized data to cloud service to re-train neural networks poses privacy concerns. Third, resource-constrained devices need to use energy-efficient communication with limited reliability and network bandwidth, which can delay or restring access of training samples to the cloud. This Chapter presents MicroTL, an approach to tackle the above challenges by tailoring Transfer Learning (TL) to resource-constrained IoT devices. MicroTL focuses on personalizing deep networks for end-users without sending sensitive IoT data to the cloud and allows IoT devices to adapt to changes in the problem domain. We evaluate MicroTL in terms of accuracy, computation, memory, and energy consumption. Notably, training with MicroTL takes 3x less energy and 2.8x less time than transmitting all data to the edge/cloud.

Paper E

Christos Profentzas, Magnus Almgren, Olaf Landsiedel

MiniLearn: On-Device Learning for Low-Power IoT Devices

Proceedings of the 22' International Conference on Embedded Wireless Systems and Networks (EWSN), 2022

CHAPTER 7

MiniLearn: On-Device Learning for Low-Power IoT Devices

Recent advances in machine learning enable new, intelligent applications in the Internet of Things. For example, today's smartwatches use Deep Neural Networks (DNNs) to detect and classify human activities. The training of DNNs, however, is done offline with previously collected and labeled datasets using extensive computational resources such as GPUs on cloud services. Once being quantized and deployed on an IoT device, a DNN commonly remains unchanged.

We argue that this static nature of trained DNNs strongly limits their flexibility to adapt to requirements that change dynamically. For example, the device may need to adjust on the fly to the limited memory and energy resources, but only the retraining or pruning of the DNN in the cloud can address these issues. Moreover, the user may need to add new classes or refine existing ones, due to different problem domains materializing dynamically. Retraining DNNs requires a high volume of data collected from IoT devices and transmitted to the cloud. However, IoT devices depend on energy-efficient communication with limited reliability and network bandwidth. In addition, cloud storage of extensive IoT data raises significant privacy concerns. This chapter introduces MiniLearn that enables re-training of DNNs on resource-constrained IoT devices. MiniLearn allows IoT devices to re-train and optimize pre-trained, quantized neural networks using IoT data collected during deployment of an IoT device. We show that MiniLearn speeds up inference by a factor of up to 2 and requires up to 50% less memory compared to original DNN. In addition, MiniLearn increases classification accuracy for a sub-set by 3% to 9% of the original DNN.

7.1 Introduction

Compression methods for Deep Neural Networks (DNNs) like quantization and pruning [9, 49] enable intelligent applications on resource-constrained

devices [9–11]. Today, smartwatches and fitness-tracker employ DNNs to detect and classify the activities of their users (Human Activity Recognition [174]) and voice-controlled virtual assistants such as Amazon's Alexa utilize DNNs on IoT devices to spot keywords [175]. However, training, optimization, and compression of DNNs are commonly conducted before deployment. The training process uses previously collected and labeled datasets and often employs vast computational resources like GPUs or even cloud services. Once deployed, DNNs commonly remain unchanged.

Should the underlying tasks change during deployment due to, for example, problem domain shifts or just the need for additional classes in a classifier, DNNs are commonly re-trained in the cloud. In addition, the device may need to adjust on the fly to the limited memory and energy resources, but only the re-training or pruning of the DNN in the cloud can address such issues. For example, when a user uses her smartwatch only for one or two activities, it would be better to act proactive and optimize the DNN for a subset of activities (and have low demand for resources) than instead disable the DNN when the battery is low, or memory is almost full.

The re-training of DNNs follows the traditional offline training approach and commonly requires uploading new datasets to cloud devices and training a new model, which is deployed on the IoT device. This is due to the following two reasons: (A) Training and refinement are both expensive in terms of memory and computation, and thus, today, avoided on resource-constrained IoT devices. Deep neural networks have millions of trainable parameters, while resource-constrained IoT devices have constrained memory, such as 64-256 KBs of RAM, run at a speed of 32-64 MHz CPU, and often operate on batteries, which rules out the training of an entire neural network from scratch. (B) Many optimization methods such as pruning [173] and quantization [21] consider deep neural network compression as the final step and make re-training of the network prohibitively hard: typically deployed low-bit, integer networks on IoT devices (i.e., 8-bit quantization) work very well for inference but face challenges such as vanishing gradient problems on gradient descent learning algorithms [21,78]. Offline training leads to significant communication overhead and often also privacy concerns, when personal data is uploaded as part of the training data to, for example, GPUs in the cloud.

In this chapter, we argue that in many application settings, the privacy-preserving way of re-training on the constrained IoT device is beneficial for the device and user, outweighing the common approach of uploading data to the cloud for training. We introduce MiniLearn, an open-source architecture training DNNs on constrained microcontrollers, filter pruning, and fine-tuning. It addresses the above challenges as follows: First, MiniLearn stores intermediate compressed outputs of quantized layer(s) as training samples to reduce memory requirements. Second, MiniLearn uses pre-trained quantized neural networks to initialize floating-point hidden layer(s) and reduces their size with static pruning. Third, after training, we quantize and fine-tune the layer(s) back to integers. A (re)trained neural network in MiniLearn consists of re-trained, quantized, and pruned layer(s).

In summary, this chapter makes the following key contributions:

- We show that low-power IoT devices can re-train and optimize pre-trained networks using data locally without the need for privacy-sensitive and

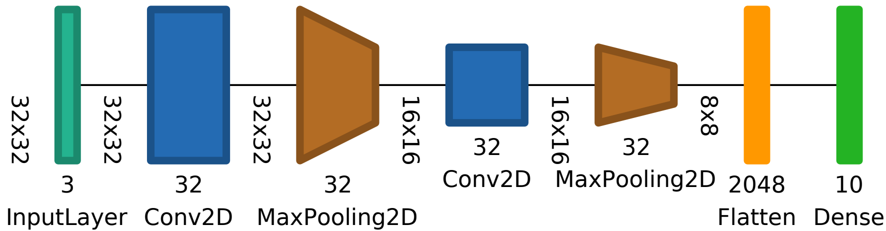


Figure 7.1: A representative architecture of a Convolution Neural Network (CNN) typically found on low-power IoT devices. The CNN consists of several layers stacking together: convolution kernels, max pooling, and fully connected (flatten & dense). The final layer is the output for the classification task.

communication intense data-upload to, i.e, cloud services for training. Moreover, we reduce memory consumption and inference latency with increased accuracy when re-training for a subset of classes.

- We design and implement MiniLearn, an open-source[1] on-device learning system for low-power IoT devices, and present the challenges and trade-offs regarding on-device learning on resource-constrained devices.

- We quantify the performance of MiniLearn in terms of accuracy, computation, memory, and energy consumption. Notably, we find that after MiniLearn, we can reduce the neural network's inference time up to 48% and memory up to 50%, with increased accuracy by 3% to 9% for a subset of the original network.

Chapter outline. The rest of the Chapter is organized as follows. Section 7.2 provides the necessary background and related work. Section 7.3 explains the main challenges for on-device learning and how MiniLearn addresses them. In Section 7.4 we describe the methodology of our experiments. In Section 7.5, we present our experimental results. In Section 7.6 we provide the related works, and Section 7.7 concludes this Chapter.

7.2 Background & Related Work

A typical deep neural network consists of the input and several hidden layers, with a final linear output providing the class prediction. While many types of neural networks exist (i.e., recurrent and convolutional), convolutional layers have prevailed since they can also solve traditional tasks of recurrent networks. For example, a problem of audio recognition can be classified using a 2D-convolutional hidden layer(s) [176], and Human Activity Recognition (HAR) can be classified using 1D-convolutional hidden layers [174]. An example of a typical Convolutional Neural Network (CNN) for IoT devices is illustrated in Figures 7.1.

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

Optimizing and compressing deep neural networks before embedded system deployment is common due to otherwise high memory, computational, and energy requirements. A common compression method is int-8 quantization [49] by using an 8-bit fixed-point representation of original floating-point values. It reduces the size without changing the original architecture of the network. Quantization typically rescales the min-max range of the neural network's weights and introduces a shifting offset for the zero-point as it may be different from real numbers after quantization. The pre-quantized network still needs to be trained in high-level libraries on high-end computers. This has led to two main variations of quantization: post-training and quantization-aware training.

Post-training quantization. With this method [9, 10, 20], the deep neural network is trained as usual in floating-point using a standard available library (e.g., Tensorflow, PyTorch). After the training period, we can use quantization statically on the network by finding a new min-max range for integer weights based on already-trained layers. This method may need a representative data set to calibrate and fine-tune the weights based on the outputs of the activation layers. This fine-tuning is a one-time process before the deployment on the IoT device.

Quantization-aware training. Quantization aware training method [21] simulates the quantization by making it part of the training process to learn the quantization values automatically. The method requires the duplication of weights for each layer in two formats. One format is in the regular 32-bit floating-point, and the other is the equivalent quantized 8-bit values. During the forward pass, it uses the integer precision layers to calculate the output of the network to simulate the deployment on the IoT device. However, during the backward pass, it uses the floating-point values to calculate the gradients and updates of the network. Then, it quantizes the floating-point weights and replaces the previous integer values. As the quantization error becomes part of the learning process, the method can lead to increased accuracy of the final quantized network. However, with this method, the quantization parameters become part of the training hyper-parameter that we need to manually tune and expect different results depending on the training process.

Pruning. Another compression method is the pruning of either the weights or filters [173, 177] of the network. The main idea is to identify and remove redundant parts of the networks, categorized in two different methods: weight pruning and filter pruning. Weight pruning [177] targets redundant zeroes and model sparsity by creating more dense layers. In contrast, filter pruning [173] considers convolutional filters as blocks and removes filters that do not contribute enough to each layer's output.

Inference. Inference with IoT devices has become widely available, for example, ARM CMSIS-NN [9] and Tensorflow Micro [20] offer libraries for quantized neural network inference on low-power devices. The libraries can optimize the neural networks for low-power inference hardware (e.g., Cortex-M, ESP), and may increase performance by utilizing available on-board DSP accelerator. However, the neural networks are trained and quantized in the cloud (typically with post-training quantization as described above).

7.3 MiniLearn Overview

This section describes the motivation behind MiniLearn. First, we introduce a motivational application scenario where we need on-device learning. Second, we present the data flow, training steps, and system challenges to enable training and fine-tuning on low-power IoT devices.

7.3.1 Application Scenario

As a motivating scenario, we consider a smartwatch application tracking users' health and fitness progress from exercise. In the past, most of the data analysis happened in the cloud, but with recent advances, we see parts of inference happen locally on the device. The smartwatch has a pre-trained network based on generic data engaging in various activities and sports. With MiniLearn, we envision a scenario where the local user shifts her activities to a smaller group of activities after using the device for a while. While the user engages in activities, the device creates a training set using the hidden layer outputs and labeling either by using another sensor or asking the user through user interaction. When the training set is filled with enough data (typically 100-600), it initiates the retraining and fine-tuning of the neural networks with appended Fully Connected (FC) layer(s). The local training set never leaves the device, and the cloud provider does not acquire any personal data, preserving the privacy of the user.

7.3.2 Data Flow

In Figure 7.2, we illustrate the flow of data collection and training process. First, a device sensor or user provides the label during the forward pass, and MiniLearn stores the pair of the label/output of the last hidden layer in a local data set. The process continues as long the device has enough storage for the training samples. In the second step, MiniLearn initiates the training process after collecting enough training data.

7.3.3 Training Steps

MiniLearn achieves on-device learning by filter pruning and learning to low-power IoT devices in three steps, illustrated in Figure 7.2. First, it utilizes efficient memory storage by collecting and storing the quantized hidden layer outputs (suitable for de-quantization) in integer format. Second, it uses floating-point hidden layers to allow training with gradient descent algorithms. Third, it de-quantizes the training data only during training, which combines a low memory footprint and adequate gradients for learning.

7.3.4 System Challenges

We identify three main system challenges when designing MiniLearn. First, common deep learning algorithms depend on a significant number of training samples that require resources unavailable on common IoT devices. Second, neural networks are typically quantized on low-power IoT for efficiency, and

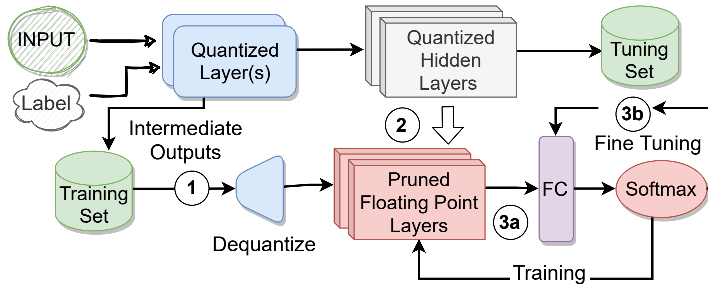


Figure 7.2: The main structure of MiniLearn and how to address the three main challenges. First, during the forward pass, it stores the output of the first layer in int-8 format, while during training, it dequantizes one batch at a time to float. Second, it dequantizes the filters to float and applies static pruning to reduce the number of trainable parameters. Third, it trains the convolutional layers and fine-tunes the Fully Connected (FC) layers.

reversing the quantization to enable additional learning on existing neural networks is not well-studied [165]. Third, even quantized neural networks have millions of parameters, and additional training requires significant resources.

(1) Training samples. First, on-device learning needs to optimize the data storage of training samples to address memory limitations, together with computational and energy trade-offs. For example, a typical batch of 32 images of the popular CIFAR-10 data set will occupy 128 KB, a significant amount for a low-power device. Typically most data sets are collected with high-dimensional features on the cloud, where data pre-processing and reduction may happen. Cloud services have an abundance of resources, and they can utilize pipelines and distributed nodes for data processing. In contrast, the local data of IoT has limited capabilities for pre-processing high-dimensional data.

(2) Learning based on quantized layers. Second, deep neural networks on low-power IoT are typically quantized to address memory and computational constraints. Quantization is straightforward and is adequate for inference-only tasks, but utilizing or reversing already quantized layers for additional learning is an open challenge [21, 78, 165]. The main reason is that after quantization, most hidden layers are restrained by the low dynamic range and precision. Since the training process of neural networks commonly uses gradient descent algorithms [78], without applying a technique to reverse or extend parts of the quantization, it leads to inadequate results [21].

(3) Number of training parameters. Third, typically deep neural networks have millions of parameters to be trained. On-device learning needs to incorporate methods to reduce the number of parameters without compromising the model's accuracy. However, standard methods (e.g., neural architecture search, pruning) for removing parameters are iterative approaches [173, 178] that find the optimal configuration through several possible architectures by storing check-points and then choosing the one performing the best. This is unfeasible using resource-constrained devices that depend on KBs of memory and batteries to operate.

7.3.5 MiniLearn Design

MiniLearn's main objective is to provide on-device learning using personalized IoT data. MiniLearn retrains neural networks towards the end-user preferences without cloud interaction by keeping the data on the device and preserving the user's privacy.

Our design is based on two key observations. First, we can reduce the training memory requirements, by dynamically collecting intermittent outputs of hidden layers in integer precision and dequantize (see bellow) to floating-point precision only during training. This approach provides a sufficient dynamic range for calculating gradients during back-propagation, avoiding duplicated weights. Second, training hidden layers in floating precision while keeping the pre-trained network in integer format reduces the demand for computational and energy resources without affecting the overall learning process.

Dequantization. To address the resource-constrained devices, we enforce the optimization and the proper placement of the dequantization process. In Figure 6.2, we present in detail where we collect the intermittent outputs and which operations are involved. We collect the output before applying ReLU to avoid consecutive zeros. Due to quantization, the pre-trained networks apply Q-shift, an offset of the difference between the integer domain's zero-point and real numbers (as explained in Section 7.2). We need to reverse both the min-max scale factors between integer and real numbers and shift back and reverse the zero-point offset. A key observation is that the division is a power of 2 which can also be done using bit-shifting, and most cross-compilers can highly optimize and combine these operations.

7.3.6 MiniLearn Architecture

In this section, we explain in detail the main parts of the MiniLearn architecture that address the three challenges stated above.

(1) Sample Storage First, MiniLearn addresses memory constraints by collecting intermediate outputs of the first layer(s) in their compressed quantized (int-8). We divide the samples into a training set for the hidden layer(s) and a tuning set for the fully connected layer(s) (see part 3a-b). The samples are restored to floating-point only during training using the scale factors from quantization.

(2) Network architecture. MiniLearn avoids dynamic range and precision problems by trading-off memory for floating-point precision for both the hidden layer(s) and fully connected layers during training. The first step is to dequantize the filters of hidden layers and prune them in case of memory constraints. We apply proper static pruning to avoid iterative approaches using standard filter ranking methods (e.g., based on the L1 - L2 norm). We let the fully connected layers compensate for the filter reduction by a final fine-tuning for a few extra epochs. Finally, we modify the fully-connected layers from the initial network to have the number of neurons corresponding to the subset we want to optimize.

(3a) Filter training. MiniLearn is able to use standard training algorithms and reduce the number of parameters with a two-step approach. We represent convolutional filters and fully connected layers with 32-bit floating-point, but we apply pruning on the filters. This way, we can train with

back-propagation and min-batch stochastic gradient descent, and at the same time reduce the number of trainable parameters. For calculating the loss of the network, we use cross-entropy. We use small batches to dequantize and temporarily store the input samples (collected from the first layer) in a 32-floating point. After training, we quantize the convolutional filters back to integer so as to address the device's memory and computational limitation during inference.

(3b) Fine-tuning. MiniLearn compensates for filter pruning quantization with a final fine-tuning on the fully connected layer(s). This step is necessary as prior pruning and quantization introduce some loss of information. The previous layers are frozen during this step, and the only trainable parameters are from the fully connected layers. The complete algorithm is presented in Algorithm 1, explained in detail in the next section. The final network consists of unchanged pre-trained layers(s), the re-trained and quantized convolutional layer(s), and floating-point fully connected layers.

7.3.7 MiniLearn Stages

In this section, we explain in detail the MiniLearn learning algorithm stage by stage. There are three main components involved: a) Pruning and filter selection, b) Re-training and adjusting the filters, and c) Final fine-tuning.

7.3.7.1 Dequantizing and Pruning of Filters (Stage-I)

We use the L2 norm of each filter-vector as a static method for selecting the convolution filters. Other options include L1 and Max norm. In the case of dequantized weights, filters with smaller L2 norm result in relatively small activations implying that they are less significant for the networks [173]. The primary purpose of pruning is to address the memory constraints of the device.

We keep intact the first convolution layer(s) in integer format to retain compressed inputs for the network. The choice of which layer to freeze and re-train is a hyperparameter of the algorithm. Next, we define a 32-bit floating-point based on the rest of the hidden convolutional layer(s) and initialize their weights by converting the quantized values back to float-point using their scale factors. Similarly, we define 32-bit floating-point fully connected layer(s) based on the pre-trained network, but we reduced the output neurons to the number of sub-classes we want to optimize. Filter pruning has a cascade effect, where pruning a prior layer reduces the input dimension of the next layer. This creates two implicit memory reductions. First, we reduce memory space from the fully connected layer(s), as they directly multiplied their weights with the input. Second, the size of intermittent hidden layer(s) results is reduced due to filter dimension reduction.

7.3.7.2 Training the Filters (Stage-II)

The second step is to train the floating-point convolutional filters using the training set collected by the IoT device. The samples consist of pre-computed outputs of the first convolution filter(s) in int-8 format. We convert each sample to a floating-point format during training per batch size. This way, we do not have to store the complete training data in floating-point. We train using the standard backpropagation algorithm with mini-batch stochastic gradient

descent. We keep a small learning rate, and we need to use a small training set in this stage. The filters are initialized based on pre-trained weights, and we only need to tune the filters for the subset of classes we want to optimize.

7.3.7.3 Fine-Tuning (Stage-III)

The final step is to quantize the pruned convolutional filters from floating-point to int-8 to utilize optimized libraries during inference. We use the scale factors used in Stage-I in reverse order to convert them back to int-8 values for each filter. This step, together with the prior pruning, introduces some loss of information. To compensate, we perform a final fine-tuning using only the fully connected layer that we keep in floating-point for the final deployment. However, we randomly initialize the weights to find a new minimum as the network weights have shifted. The training algorithm is the same as Stage-II, with the difference that we train only the fully connected layer with the fine-tuning set, and we forward-pass through the new weights of the convolutional filters. The final network consists of the pruned and quantized convolutional filters with the newly trained fully connected layer(s).

7.4 Experimental Methodology

In this section, we describe the methodology of our experiments. Connecting with the application scenario in Section 7.3, the goal of the evaluation is to show the trade-offs of optimizing a pre-trained neural network (e.g., activity recognition with a smartwatch) to a sub-set of classes (two or three activities) and reducing the demand for resources of the IoT device. We start with software and hardware implementation, and then we present the datasets, baselines, and pre-trained network architectures.

Software & hardware setup. We use deep neural networks pre-trained in PyTorch and Python 3.8. We use the PyTorch MinMax Quantization to extract the 8-bit integer weights. We implement MiniLearn in C, and we compile our code using the Arm-GCC 9.2.1. for Cortex-M hardware. Finally, we utilize the CMSIS-NN [9] library for inference on IoT devices. We evaluate MiniLearn on nRF-52840 SoC featuring: a 32-bit ARM Cortex-M4 with FPU at 64 MHz, 256 KB of RAM, and 1 MB of Flash. We use the Nordic-Semiconductors Power Profiler Kit (PPK) shield [167] to accurately measure the power consumption.

Data sets. We use three representative datasets, one for audio recognition, one for color-image recognition, and one for human activity recognition.

CIFAR-10: A widely used image classification dataset. The dataset consists of 60,000 32x32 color images, where 50,000 are for training and 10,000 for testing. There are ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

Google Keyword Spotting (KWS) [15] (v2): The dataset contains 110,000 audio samples with a total of 35 classes. Each audio sample is a keyword recorded as a speech command of a small duration. We make the following common pre-processing to create images from audio files [176]. We generate 2D images by taking the time window of the audio (timestamp) as the width and Mel-frequency cepstral coefficients (MFCC [176]) as height. The window

size is 31.25 ms with a moving step at 16.125 ms. The data set is divided into 85,000 training samples and 11,000 test samples in total.

WISDM-HAR. Wireless Sensor Data Mining (WISDM) [179] is a human activity recognition data set. The data are collected and labeled by using accelerometers of Android-based cell phones. It has six classes (activities) stored as time series: 1) walking, 2) jogging, 3) walking upstairs, 4) walking downstairs, 5) sitting, 6) standing. The data type is a time series of signals needing pre-processing before use. Our experiments parse the data per 128 time-step and apply min-max normalization over the nine-axis (final dimension 128x9), leading to 7,300 training samples and 3,000 testing samples.

Class subsets. We generate subsets from the initial classification domains for our experiments as follows. For CIFAR-10, we reduce from 10 to 3 classes. For KWS, we reduce from 35 to 5 classes. For WISDM-HAR, we reduce from 6 to 3 classes. The number of classes reflects the memory constraints of the device; for more classes, we will need to extend the device's hardware capabilities. We kept data samples (100 - 600) data separate while training the baseline network and we use the separate data for retraining and fine-tuning with MiniLearn. We repeat our experiments over 30 times, randomly drawing a subset of unique (with no replacement) 3-classes for CIFAR, 3-classes for WISDM-HAR, and 5-classes for KWS subsets, respectively. The training sets on the device (MiniLearn) are used by 10% for filter training and 90% for fine-tuning across all experiments. We report the average accuracy as percent (%) over the test-set and standard deviation as $\pm$.

Baselines. We compare the accuracy of MiniLearn on IoT devices with two baselines. a) Accuracy we would have obtained if we used the **Original** pre-trained network tested on the subset classes. This baseline represents the current approach of pre-deployed neural networks on IoT devices without learning capabilities. b) Accuracy we would have obtained if we retrain and download a new model from the cloud services optimized for the 3-class and 5-class subset, respectively. This *cloud* baseline represents the ideal case where the cloud has access to previous models and as well all-new personal IoT data sent by the device, and it acts as an upper bound, as arguably, the cloud service has access to a wide range of training and optimization methods.

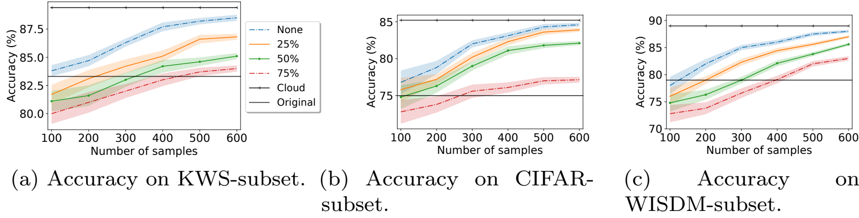
Pre-trained neural networks Our experiments use three pre-trained and quantized neural networks [176] reported in Table 7.1. We report the shape size for each layer: Convolutional 1D & 2D, MaxPooling, and Fully Connected. We also report the Multiply-Accumulate operations (MAC) per cycle (in Millions) and the size of the layer output in Bytes.

Data & code availability. We provide the source code and evaluation data of MiniLearn in a public repository.[2]

7.5 Results & Discussion

This section presents the results of our MiniLearn evaluation on low-power IoT devices. We answer the following questions. (a) Is it feasible to re-train and improve existing deployed deep learning networks with on-device learning? (b) What is the performance of on-device learning applied to low-power devices?

[2]https://github.com/chrpro/MiniLearn



(a) Accuracy on KWS-subset. (b) Accuracy on CIFAR-subset. (c) Accuracy on WISDM-subset.

Figure 7.3: Average accuracy using MiniLearn with different pruning percentages on KWS 7.3a, CIFAR 7.3b, and WISDM 7.3c. The original baseline is the pre-trained network test on the sub-set, and *cloud* represents sending data and retraining in the cloud. The shaded area represents the standard deviation.

(c) What is the overhead of MiniLearn in terms of computation, memory, and energy consumption?

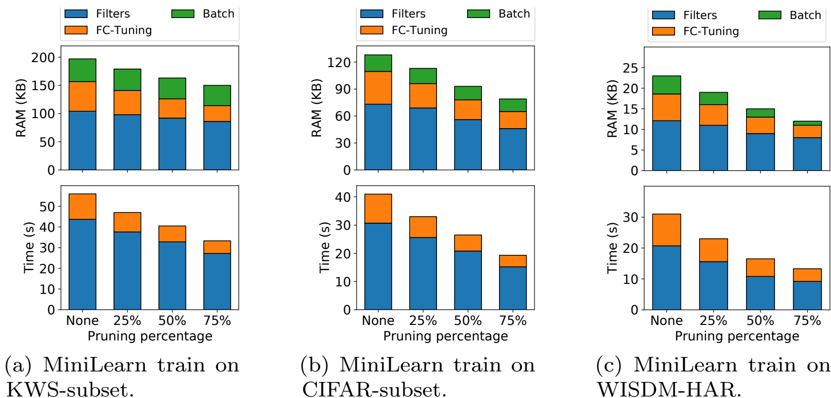
7.5.1 MiniLearn Accuracy Results

In Figure 7.3, we compare the average accuracy of the baselines **Cloud** and **Original** to **MiniLearn** using different percentages of filter pruning with the different number of training samples. We prune by using a 25% step for each experiment, while the architecture of **Cloud** and **Original** are unchanged. None refers to no pruning at all.

In Figure 7.3a, we present the accuracy of the 5-subclasses on the KWS with different pruning percentages. In Figure 7.3b, we present the accuracy of the 3-subclasses from the CIFAR-10, and in Figure 7.3c, we present the accuracy of the 3-subclasses on WISDM-HAR. We observe that without pruning, we can immediately increase the accuracy on the device and reach equivalent accuracy with the **Cloud**, by using 600 samples locally instead of retraining and downloading a new model from the Cloud. With aggressive pruning (e.g., 75%), even though we can save significant memory and energy for the device, the capacity of the network decreases, and with 100 samples, the accuracy is less than the original one. However, increasing the training samples to 400 and 500, respectively, even the pruned neural networks have better accuracy than the original. We observe a similar trend for all data sets in Figure 7.3. However, we need more training samples on the KWS date set to compensate for the loss of accuracy due to the pruning compared to the CIFAR-subset and WISDM-subset. The main reason is that KWS-subset has more classes than the other data sets.

Communication Cost. For the approach of the cloud-based baseline, we calculate the communication cost for sending training samples from the device to the cloud by utilizing short-range Low-Power Bluetooth (BLE) commonly available on low-power IoT devices. We report the transmission time using the standard throughput of 700 kbps (payload) using the default 1 Mbps mode [166]. For the energy communication cost, we report the energy required by BLE [166] using the PPKv1. by applying 3 V, which is 7.5 mA.

With the cloud baseline, in order to train a new neural network optimized for the sub-set classes, we need to upload training samples from the device



(a) MiniLearn train on KWS-subset.

(b) MiniLearn train on CIFAR-subset.

(c) MiniLearn train on WISDM-HAR.

Figure 7.4: MiniLearn performance during training in terms of time and memory consumption (using 600 samples). We apply different pruning percentages with a 25% step.

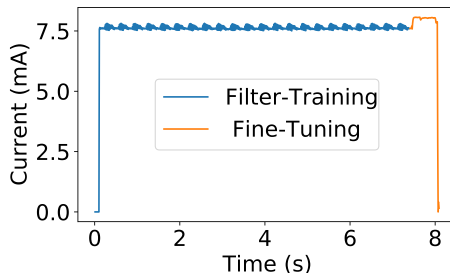
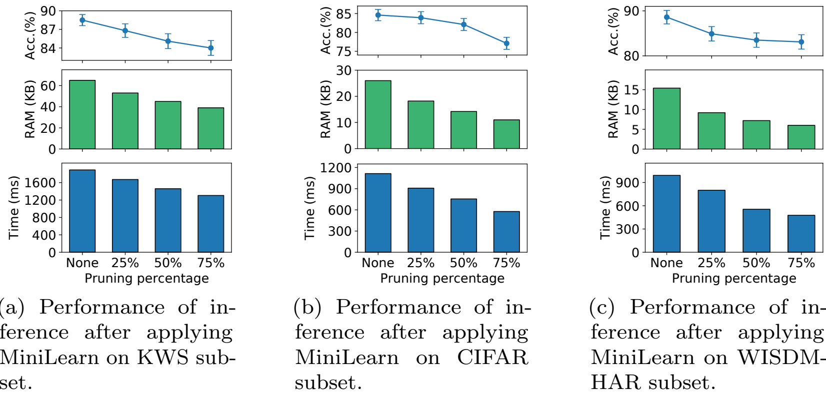


Figure 7.5: Electric current drawn during training on nRF-52840-DK, by applying at 3 V, during a complete training with 100 samples on CIFAR-subset.

to the cloud. After training and optimizing a new neural network, we need to download the model from the cloud to the device. With 600 samples, the device needs to send to the cloud a total of 446 KB personal data and download a network update of 45 KB, for the KWS sub-set. For the CIFAR sub-set, it is 1,842 KB for personal data and 96 KB for the network. For the WISDM sub-set, it is 162 KB for personal data and 18 KB for the network. Using BLE communication, it will take 2 seconds and consume 45 mJ for the WISDM sub-set, 22 seconds and consume 462 mJ for CIFAR-3, 5,6 seconds and consumes 210 mJ for KWS. Even though sending the data to the cloud and downloading a new model will consume less energy than retraining with MiniLearn (See Table 7.2), the difference is not significant to justify the sacrifice of user's privacy with the cloud approach.

7.5.2 MiniLearn Performance on IoT Devices

In this part, we present the performance in terms of memory, computational, and energy consumption. We repeat each experiment 30 times, and we report



(a) Performance of inference after applying MiniLearn on KWS subset.

(b) Performance of inference after applying MiniLearn on CIFAR subset.

(c) Performance of inference after applying MiniLearn on WISDM-HAR subset.

Figure 7.6: Inference time and memory consumption of the neural network after applying MiniLearn with corresponding accuracy (trained on 600 samples). We apply different pruning percentages with a 25% step.

the following results: The memory allocation for RAM and Flash on nRF-52840 SoC. The average inference time is based on the cycle clock register. The average energy consumption, based on the average electric current draw in mA, by applying 3 V, and we report the standard variation as ±, when significant.

Memory Footprint. We present two aspects of the memory consumption on the device using MiniLearn. First, we report the consumption that is needed to retrain and optimize the neural network. We use flash memory only during training to store and read the training sets. In Table 7.2, we see the flash memory consumption. MiniLearn needs only a small batch of data, and each epoch reads them from the flash memory. For a sample size of 100, flash consumption is 42% with KWS, 23% with CIFAR, and 6% with WISDM-HAR However, for a sample size of 600, the consumption reaches 92% with KWS, 83% with CIFAR, and 20% with WISDM-HAR.

Figure 7.4 illustrates the parts of the RAM being used: a) The constant-size training batch (de-quantized each epoch) is read from the flash. b) Memory for the training of the convolutional filters. For example, on KWS-subset, the memory reaches 196 KB of RAM without pruning, while on CIFAR-subset reaches 128 KB of RAM, and on WISDM-subset reaches 92 KB. Each percentage of pruning reduces the RAM consumption accordingly. c) Memory for fine-tuning the fully connected layer(s). Fine-tuning takes significantly smaller memory than the overall training. For example, without pruning, on KWS-subset, fine-tuning takes 12 KB , while on CIFAR-subset takes 10 KB, and 6 KB on WISDM-HAR-subset.

Second, we report the memory consumption we save after retraining with MiniLearn during inference. In Figure 7.6, we illustrate the memory consumption during inference of neural networks with different pruning optimization of the network. We illustrate the RAM consumption with the corresponding accuracy after MiniLearn using 600 samples. The original memory consumption of the KWS network is 61 KB, the CIFAR network is 23 KB, and WISDM-HAR is 12 KB. With progressive pruning (75%), the KWS network consumption

is reduced to 31 KB, the CIFAR network is reduced to 11.6 KB, and the WISDM-HAR network is reduced to 6 KB, which is almost 50% less memory than the original, while the accuracy has slightly increased compared to the original network (see Figure 7.3).

Performance. We present two aspects of the device's performance. First, we report the time to retrain and optimize the neural network using MiniLearn. In Table 7.2, we report the performance during complete training (including fine-tuning) with different data sizes and no pruning, while in Figure 7.4, with different percentages of pruning using 600 samples. The training time is significantly reduced as pruning removes training parameters. The time for fine-tuning is reduced with higher degrees of pruning due to the cascade effect: the number of neurons for the fully connected layers is reduced with smaller filters. For example, on CIFAR-subset, applying 75% pruning MiniLearn takes 45% less time to retrain the network.

Second, we report the time we save after retraining the network, measuring the time to perform inference. In Figure 7.6, we illustrate the inference time after applying retraining with MiniLearn. The initial inference time of the KWS network (with no pruning) takes 1890 ms, the CIFAR network takes 1110 ms, and the WISDM-HAR takes 992 ms. However, with 75% pruning, the KWS network takes 945 ms, the CIFAR network takes 577 ms, and the WISDM-HAR takes 477 ms, respectively, which is almost 48% less time compared to the original.

Energy Consumption. In Figure 7.5, we illustrate the electric current draw during training with 100 samples on CIFAR-subset. The average current is 7.6 mA, drawn mainly by the FPU unit. Compared to inference using CMSIS-NN, the average current is 8 mA drawn mainly by the DSP unit. In Table 7.2, we report the energy consumption of MiniLearn without pruning and data sizes from 100 to 600. Energy consumption is related to the training time. As we increase the data size from 100 to 200, the algorithm takes more time which consumes more energy. For example, to achieve an accuracy of 88.5% close to **Cloud** baseline on the KWS-subset, we need to train with 600 samples and spend 1486 mJ.

7.5.3 Discussion and Limitations

In this section, we provide an overview of the remaining challenges of on-device learning for low-power devices and discuss the main limitations of our approach.

Number of classes. With MiniLearn, we can reduce the number of classes on the fly for resource efficiency and personalization of the neural network on the end-user. However, due to resource constraints, we can only apply MiniLearn for a subset of the original classes. If we want to increase the number of classes or personalize the network to more classes for the end user, we need to increase the memory and energy capabilities of the IoT device.

Convolutions filters. A major trade-off we face is the number of convolutions filters we want to prune. The neural network needs to have an adequate number of filters to parse the input data and extract significant features. However, the number of filters we can train with MiniLearn is limited by the memory of the device.

Energy. It is not necessary to apply pruning with MiniLearn if we want

to increase the accuracy for a particular subset of classes. However, without pruning, MiniLearn requires an increase in energy consumption for retraining the filters. We can reduce the energy consumption for training by applying pruning, but this will reduce the accuracy, indicating a significant trade-off between energy and accuracy.

Privacy. Besides reducing the dependency on cloud services, which usually have a subscription, one of the main benefits of MiniLearn is that we keep the user's personal data on the device. If privacy and communication reliability are not such issues, the cloud will be preferable to optimize the neural network.

7.6 Related Work

This section lists the related work starting with previous work on quantization and pruning, which are the primary methods for compressing DNNs. Next, we present other work on-device for edge devices compared to our method for low-power devices. The related work includes edge learning and federating learning. Finally, we present other methods that do not consider learning on the device.

Quantization Previous works focused on quantization [49] assume the process takes place in the cloud, and disregard any further optimization after deployment on the device. Contrary, MiniLearn utilizes on-device learning using data available after deployment. On the other hand, training with quantized networks [78] or using so-called quantize-aware training [21], are not applicable for low-power devices as they assume powerful devices, for example, storing both the floating-point and integer representation of the network.

Pruning Pruning on convolutional filters [173] has been shown to be computationally more efficient than targeting the overall network. One of the reasons is that in large neural networks, the fully connected layer occupies most of the space, while the convolutional filters [177, 180] require repetitive computations. However, current pruning methods are part of cloud training without targeting learning opportunities with on-device learning. In contrast, MiniLearn targets low-power IoT devices for learning opportunities on existing deployed DNN without cloud interaction.

On-device learning. The majority of Machine Learning (ML) methods primarily occur in cloud services, but recently we see a shift from the classic cloud-based training and inference in the cloud towards collecting data and training on edge devices. One example is TensorFlow Lite [163] which utilizes smartphones and other edge devices for machine learning training close to the end-user. However, other proposals exist for on-device and online learning in the literature. To start with, incremental learning [170], transfer learning [51], and few-shot learning [172] are some alternatives. With these methods, the existing models may retrain and add new classes by using a small number of training samples. TinyOL [181] utilizes online learning using encoder-decoders. However, TinyOL never reduces the size of the network and uses mean square error (MSE) as a loss function, commonly used for regression problems. MiniLearn proposes an architecture for on-device training of convolutional networks using the cross-entropy loss function suitable for classification problems, also reducing the size of the network.

Federated learning. A distributed version of on-device learning is Federated Learning (FL). With FL, multiple devices participate in training a global model. Each device uses its own datasets and needs to exchange only the training parameters with each other, for example, in FedHome [85] and FedHealth [3]. In order to ensure the privacy of the users, federated learning needs to make heavy use of cryptographic protocols such as Secure Multiparty Computation (SMC) [87] and Homomorphic Encryption (HE) [86]. Currently, federated learning is available only on edge devices. MiniLearn complements federating learning by bringing parts of training on low-power devices. Federated learning can utilize MiniLearn after creating a global model to personalize the model for the end-user using her device only.

Multi-task Learning. Another method to alter classes on the fly without retraining the complete neural network is multi-task learning [182]. Another concept to optimize multi-task learning is the virtualization of weight parameters to share weights of a neural network across multiple networks in the device [183]. However, low-power IoT devices have extreme memory constraints, and in order to benefit, they need to be equipped with a RAM capacity of MBs, compared to KBs of RAM typically found on low-power devices. In contrast, MiniLearn has an immediate optimization on a low-power device with memory and computational constraints.

Network architecture search. Another method typically used by cloud services in network architecture search NAS [77, 184]. With NAS, the designing and training of optimized neural networks are automated for the specific hardware and application domain. NAS tries to search the space of possible DNN architecture with a search strategy and performance goal for the target hardware or application. NAS provides a static optimization on an estimation of the device's resource utilization, while MiniLearn compliments NAS in that it can dynamically address changes in computational, memory, and energy requirements.

Offloading. Methods that do not utilizing re-training of DNNs focus on partitions of networks between a IoT and edge minimize execution latency [185], but increase the communication cost. Similarly, distributed inference hierarchies can offload inputs between cloud, edge, and IoT devices [186]. Adapting these methods on the fly for the IoT device will need several updates with high communication costs. Other frameworks [187] use offloading for forward propagation in DNNs with a focus on battery energy optimization for mobile devices. MiniLearn differs by focusing on learning opportunities on the device and avoiding cloud interaction.

7.7 Conclusion

This Chapter proposes MiniLearn to re-train and improve pre-trained neural networks on resource-constrained IoT devices. MiniLearn improves the accuracy on a subset of classes using local data, and it can reduce the memory and inference latency of the initial network using filter pruning and fine-tuning. Evaluation results on real embedded hardware demonstrate that after MiniLearn, pre-trained neural networks can take 45% less inference time and 50% less memory, increasing at the same time the accuracy for a classification sub-set

by 3% to 9%.

Algorithm 3: MiniLearn Algorithm

```
   Data: Training Set, Fine-tuning Set
1  Stage-I:Pruning
2  for Convolutional-layer in Network do
3      for Filter in Convolutional-layer do
4          FilterScores[i] ¡- Norm L2 (Filter) ; //select filters
5      end
6      Pruning (Percentage, FilterScores)
7  end
8  Stage-II: Training Filters
9  for epoch e = 1, ... , E do
10     for Sample in Training Set do
11         // 32-bit floating point format
12         Y^(i), X^(i) := de-quantize (Sample, Q-factors);
13         Y^(j) := Forward (Hidden-layers, X^(i));
14         delta := - sum_{i=1}^{m} y_i log(y_j) // cross entropy loss
15         Back propagation (Hidden-layers, delta);
16     end
17 end
18 for Filter in Convolutional-layer do
19     Quantize (Filter, Q-factors) ; //quantize filters back to int
20 end
21 Stage-III: Fine-Tuning
22 FC := Randomized Fully Connected Layer(s)
23 Freeze (Hidden-Layers) // no updates prior to FC
24 for epoch e = 1, ... , E do
25     for batch b in Fine-Tuning Set do
26         // 32-bit floating point format
27         Y^(i), X^(i) := de-quantize (b, Q-factors);
28         Y^(j) := Forward (FC, X^(i));
29         delta := - sum_{i=1}^{m} y_i log(y_j) // cross entropy loss
30         Back propagation (FC, delta);
31     end
32 end
```

Table 7.1: Networks used in the experiments. The table shows the shape size, computations in terms of multiply-accumulate (MAC), and the layer output in Bytes for each layer.

Pre-trained Network on KWS

Layer	Shape	MAC	Output
Input	(62, 12, 1)	-	744
Conv1D	(58, 8, 16)	0.19M	7,424
MaxPool	(29, 8, 16)	0.06M	3,712
Conv1D	(27, 6, 32)	0.75M	5,184
MaxPool	(13, 6, 32)	0.03M	2,496
Conv1D	(11, 4, 64)	0.81M	2,816
Conv1D	(9, 2, 32)	0.33M	576
FC	(576, 35)	0.03M	35

Pre-trained Network on CIFAR-10

Layer	Shape	MAC	Output
Input	(32, 32, 3)	-	3,072
Conv2D	(30, 30, 32)	1.55M	28,800
MaxPool	(15, 15, 32)	0.07M	7,200
Conv2D	(13, 13, 32)	1.11M	5,408
MaxPool	(6, 6, 32)	0.02M	1,152
Conv2D	(4, 4, 32)	0.73M	512
FC	(512 , 10)	0.03M	10

Pre-trained Network on WISDM

Layer	Shape	MAC	Output
Input	(90, 3, 1)	-	270
Conv1D	(90, 32 ,1)	0.03	2,880
Conv1D	(90, 32 , 1)	0.03M	2,880
Conv1D	(90, 32 ,1)	0.03M	2,880
FC	(96, 128)	0.03M	12,288
FC	(128, 6)	0.03M	6

Table 7.2: MiniLearn training evaluation, without pruning, on CIFAR and KWS subsets. We report the RAM and Flash memory footprints, the average training time, and average energy consumption during the complete training (including fine-tuning).

MiniLearn on KWS-subset.

Data-Size	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)	Accuracy (%)
100	196	439	10±0.2	254±0.2	83.8±0.3
200	196	580	16±0.1	406±0.1	84.7±0.2
300	196	620	25±0.1	635±0.1	86.3±0.2
400	196	761	36±0.1	914±0.1	87.7±0.2
500	196	840	47±0.1	1194±0.1	88.2±0.1
600	196	950	56±0.1	1486±0.1	88.5±0.1

MiniLearn on CIFAR-subset.

Data-Size	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)	Accuracy (%)
100	128	240	8±0.2	203±0.2	76.8±0.5
200	128	370	16±0.2	406±0.2	78.8±0.3
300	128	510	23±0.2	584±0.2	82.1±0.2
400	128	660	30±0.2	762±0.2	83.1±0.2
500	128	790	35±0.2	890±0.2	84.3±0.1
600	128	857	40±0.2	1016±0.2	84.6±0.1

MiniLearn on WISDM-HAR subset.

Data-Size	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)	Accuracy (%)
100	92	64	6±0.1	138±0.2	78.1±0.3
200	92	91	10±0.1	228±0.1	82.1±0.2
300	92	118	14±0.1	319±0.1	85.3±0.2
400	92	145	21±0.1	479±0.1	87.7±0.2
500	92	172	25±0.2	670±0.1	88.5±0.1
600	92	201	30±0.3	786±0.1	89.5±0.1

Bibliography

[1] 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.

[2] 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.

[3] Y. Chen, X. Qin, J. Wang, C. Yu, and W. Gao, "Fedhealth: A federated transfer learning framework for wearable healthcare," *IEEE Intelligent Systems*, vol. 35, no. 4, pp. 83–93, 2020.

[4] L. R. Suzuki, "Smart cities iot: Enablers and technology road map," in *Smart City Networks*. Springer, 2017, pp. 167–190.

[5] 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.

[6] 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.

[7] S. Biswas, K. Sharif, F. Li, B. Nour, and Y. Wang, "A scalable blockchain framework for secure transactions in iot," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4650–4659, 2018.

[8] 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.

[9] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," in *arXiv:1801.06601*, 2018.

[10] J. Lin, W. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "Mcunet: Tiny deep learning on iot devices," in *Advances in Neural Information Processing Systems, NeurIPS*, 2020.

[11] C. Profentzas, M. Almgren, and O. Landsiedel, "Performance of deep neural networks on low-power iot devices," in *CPS-IoTBench '21*. Association for Computing Machinery, 2021.

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

[13] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones," in *European Symposium on Artificial Neural Networks (ESANN)*, 2013.

[14] A. S. Razavian, H. Azizpour, J. Sullivan *et al.*, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014.

[15] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," *arXiv:1804.03209*, 2018.

[16] A. T. Manual, "Arm ethos-u55 npu technical reference manual," 2021.

[17] N. P. Jouppi, C. Young, and E. al, "In-datacenter performance analysis of a tensor processing unit," in *arXiv:1704.04760*, 2017.

[18] A. L. team, "Arm trustzone," https://www.arm.com/products/security-on-arm/trustzone, 2017.

[19] A. S. D. Manual, "Arm cryptocell-312 runtime software developers manual," 2019.

[20] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "Tensorflow lite micro: Embedded machine learning on tinyml systems," *arXiv:2010.08678*, 2021.

[21] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[22] W. Fang, F. Xue, Y. Ding, N. Xiong, and V. C. M. Leung, "Edgeke: An on-demand deep learning iot system for cognitive big data on industrial edge devices," *IEEE Transactions on Industrial Informatics*, 2021.

[23] 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.

[24] 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.

[25] 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.

[26] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

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

[28] R. C. Merkle, “A Digital Signature Based on a Conventional Encryption Function,” in *Advances in Cryptology - CRYPTO ’87, Conference on the Theory and Applications of Cryptographic Techniques*, 1987.

[29] 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.

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

[31] 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.

[32] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, “Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs,” in *Cryptographic Hardware and Embedded Systems - CHES 2004*, M. Joye and J.-J. Quisquater, Eds. Springer Berlin Heidelberg, 2004.

[33] J. Poon and V. Buterin, “Plasma: Scalable autonomous smart contracts,” *Plasma.io*, 2017.

[34] 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.

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

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

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

[38] Y. Ye, J. Zhang, W. Wu, X. Luo, and J. Cao, "Boros: Secure Cross-Channel Transfers via Channel Hub," *arXiv:1911.12929*, 2019.

[39] 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.

[40] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016.

[41] K. Bong, S. Choi, C. Kim, S. Kang, Y. Kim, and H. Yoo, "14.6 a 0.62mw ultra-low-power convolutional-neural-network face-recognition processor and a cis integrated with always-on haar-like face detector," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2017.

[42] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[43] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning.* PMLR, 2013, pp. 1139–1147.

[44] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research*, vol. 12, no. 7, 2011.

[45] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[46] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," *CoRR*, vol. abs/1711.05101, 2017. [Online]. Available: http://arxiv.org/abs/1711.05101

[47] F. Girosi, M. Jones, and T. Poggio, "Regularization theory and neural networks architectures," *Neural computation*, vol. 7, no. 2, pp. 219–269, 1995.

[48] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[49] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research. PMLR, 2019.

[50] R. David, J. Duke, A. Jain, V. J. Reddi *et al.*, "Tensorflow lite micro: Embedded machine learning on tinyml systems," 2020.

[51] S. J. Pan, J. T. Kwok, Q. Yang *et al.*, "Transfer learning via dimensionality reduction." in *AAAI*, 2008.

[52] H. Zhu, J. Xu, S. Liu, and Y. Jin, "Federated learning on non-iid data: A survey," *arXiv:2106.06843*, 2021.

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

[54] Z. Bao, W. Shi, D. He, and K.-K. R. Chood, "IoTChain: A Three-Tier Blockchain-based IoT Security Architecture," *arXiv:1806.02008 [cs]*, 2018.

[55] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards Blockchain-based Auditable Storage and Sharing of IoT Data," in *Proceedings of Cloud Computing Security Workshop - CCSW*. ACM Press, 2017.

[56] 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.

[57] 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.

[58] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. Vienna, Austria: ACM Press, 2016.

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

[60] M. Samaniego and R. Deters, "Internet of Smart Things - IoST: Using Blockchain and CLIPS to Make Things Autonomous," in *2017 IEEE International Conference on Cognitive Computing (ICCC)*. Honolulu, HI, USA: IEEE, 2017.

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

[62] L. Eckey, S. Faust, and B. Schlosser, "Optiswap: Fast optimistic fair exchange," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 543–557.

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

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

[65] R. Network, “What is the raiden network?” *https://raiden .network/*, 2018.

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

[67] 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.

[68] 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.

[69] 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.

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

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

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

[73] P. Chatzigiannis, F. Baldimtsi, C. Kolias, and A. Stavrou, “Black-box iot: Authentication and distributed storage of iot data from constrained sensors,” in *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, ser. IoTDI ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–14. [Online]. Available: https://doi.org/10.1145/3450268.3453536

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

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

[76] A. Kurt, S. Mercan, E. Erdin, and K. Akkaya, “3-of-3 multisignature approach for enabling lightning network micro-payments on iot devices,” *CoRR*, vol. abs/2109.09950, 2021. [Online]. Available: https://arxiv.org/abs/2109.09950

[77] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, “Efficient neural architecture search via parameters sharing,” in *Proceedings of the 35th International Conference on Machine Learning*. PMLR, 2018.

[78] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[79] C. Leng, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with admm," 2017.

[80] Q. Chen, Z. Zheng, C. Hu, D. Wang, and F. Liu, "Data-driven task allocation for multi-task transfer learning on the edge," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[81] Q. Sun, Y. Liu, T.-S. Chua, and B. Schiele, "Meta-transfer learning for few-shot learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[82] H. Cai, C. Gan, L. Zhu, and S. Han, "Tinytl: Reduce memory, not parameters for efficient on-device learning," in *arXiv:2007.11622*, 2021.

[83] D. Li, T. Salonidis, N. V. Desai, and M. C. Chuah, "Deepcham: Collaborative edge-mediated adaptive deep learning for mobile object recognition," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[84] J. Kim, Y. Bhalgat, J. Lee, C. Patel, and N. Kwak, "Qkd: Quantization-aware knowledge distillation," *arXiv:1911.12491*, 2019.

[85] Q. Wu, X. Chen, Z. Zhou, and J. Zhang, "Fedhome: Cloud-edge based personalized federated learning for in-home health monitoring," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2020.

[86] T. Graepel, K. Lauter, and M. Naehrig, "Ml confidential: Machine learning on encrypted data," in *Information Security and Cryptology (ICISC) 2012*. Springer, 2013.

[87] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "Ezpc: Programmable and efficient secure two-party computation for machine learning," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019.

[88] H.-J. Jeong, H.-J. Lee, K. Yong Shin, Y. Hwan Yoo, and S.-M. Moon, "Perdnn: Offloading deep neural network computations to pervasive edge servers," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020.

[89] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[90] M. Major, *A Taxonomic Evaluation of Rootkit Deployment, Behavior and Detection*. ProQuest Dissertations Publishing, 2015. [Online]. Available: https://books.google.se/books?id=UQ1YAQAACAAJ

[91] B. Grill, A. Bacs, C. Platzer, and H. Bos, *"Nice Boots!" - A Large-Scale Analysis of Bootkits and New Ways to Stop Them.* Cham: Springer International Publishing, 2015, pp. 25–45. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-20550-2_2

[92] A. S. Kushwaha, "A trusted bootstrapping scheme using usb key based on uefi," *International Journal of Computer and Communication Engineering*, vol. 2, no. 5, pp. 543–546, 09 2013, copyright - Copyright IACSIT Press Sep 2013; Last updated - 2014-04-11. [Online]. Available: http://proxy.lib.chalmers.se/login?url=http://search.proquest.com.proxy.lib.chalmers.se/docview/1439562246?accountid=10041

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

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

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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.

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

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

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

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

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

[108] 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.

[109] 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.

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

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

[112] 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.

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

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] 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.

[119] 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.

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

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

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

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

[124] 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.

[125] 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.

[126] 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.

[127] 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.

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

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

[130] 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.

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

[132] 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.

[133] 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.

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

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

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

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

[138] 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.

[139] 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

[140] 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.

[141] 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.

[142] 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.

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

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

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

[146] 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.

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

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

[149] 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.

[150] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *International Conference on Learning Representations (ICLR)*, 2020.

[151] J. B. Yang, M. N. Nguyen, P. P. San *et al.*, "Deep convolutional neural networks on multichannel time series for human activity recognition," in *AAAI Press*, 2015.

[152] I. Mehmood, A. Ullah, K. Muhammad *et al.*, "Efficient image recognition and retrieval on iot-assisted energy-constrained platforms from big data repositories," in *IEEE Internet of Things Journal*, 2019.

[153] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[154] C. Wu, D. Brooks, K. Chen *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[155] "uTensor tinyml ai inference library," 2019, https://utensor. github.io/.

[156] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[157] J. Yang, X. Shen, J. Xing *et al.*, "Quantization networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[158] "STM32Cube ai," 2020, https://www.st.com/en/ embedded -software/x-cube-ai.html.

[159] N. Rotem, J. Fix, S. Abdulrasool *et al.*, "Glow: Graph lowering compiler techniques for neural networks," 2018.

[160] V. J. Reddi, C. Cheng, D. Kanter *et al.*, "Mlperf inference benchmark," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[161] C. Coleman, D. Narayanan, D. Kang *et al.*, "Dawnbench: An end-to-end deep learning benchmark and competition," *Conference on Neural Information Processing Systems (NIPS)*, 2017.

[162] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[163] "Tensorflow methods," 2021, https://www.tensorflow.org/.

[164] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? Adaptive rounding for post-training quantization," in *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 2020.

[165] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Hawq: Hessian aware quantization of neural networks with mixed-precision," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[166] "Nordic Semiconductor technical documentation," 2019, https://infocenter.nordicsemi.com/.

[167] "Nordic Semiconductorpower profiler kit," 2019, https://www.nordicsemi.com/Software-and-tools/Development-Tools/Power-Profiler-Kit.

[168] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[169] A. Anjomshoaa and E. Curry, "A transfer learning framework for iot-enabled environments," in *International Conference on Internet of Things Design and Implementation (IoTDI) '21*. ACM, 2021.

[170] J. Bai, A. Yuan, Z. Xiao, H. Zhou, D. Wang, H. Jiang, and L. Jiao, "Class incremental learning with few-shots based on linear programming for hyperspectral image classification," *IEEE Transactions on Cybernetics*, 2020.

[171] Y. Chen, Y. Ma, T. Ko, J. Wang, and Q. Li, "Metamix: Improved meta-learning with interpolation-based consistency regularization," in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021.

[172] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. Torr, and T. M. Hospedales, "Learning to compare: Relation network for few-shot learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[173] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv:1608.08710*, 2016.

[174] S.-M. Lee, S. M. Yoon, and H. Cho, "Human activity recognition from accelerometer data using convolutional neural network," in *IEEE International Conference on Big Data and Smart Computing*, ser. BigComp '17, 2017.

[175] H. Chung, M. Iorga, J. Voas, and S. Lee, ““alexa, can i trust you?”,” *IEEE Computer Journal*, vol. 50, no. 9, pp. 100–104, 2017.

[176] S. K. Gouda, S. Kanetkar, D. Harrison, and M. K. Warmuth, “Speech recognition: Keyword spotting through image recognition,” in *arXiv:1803.03759*, 2020.

[177] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. Doermann, “Towards optimal structured cnn pruning via generative adversarial learning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[178] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, “Progressive neural architecture search,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[179] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, “Activity recognition using cell phone accelerometers,” in *Proceedings of the Fourth International Workshop on Knowledge Discovery from Sensor Data*, 2010, pp. 10–18.

[180] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *International Conference on Learning Representations (ICLR)*, 2016.

[181] H. Ren, D. Anicic, and T. Runkler, “Tinyol: Tinyml with online-learning on microcontrollers,” *arXiv*, 2021.

[182] S.-G. Leem, I.-C. Yoo, and D. Yook, “Multitask learning of deep neural network-based keyword spotting for iot devices,” *IEEE Transactions on Consumer Electronics*, vol. 65, no. 2, pp. 188–194, 2019.

[183] S. Lee and S. Nirjon, “Fast and scalable in-memory deep multitask learning via neural weight virtualization,” in *ACM International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’20. Association for Computing Machinery, 2020.

[184] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, “Efficient architecture search by network transformation,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[185] X. Chen, M. Li, H. Zhong, Y. Ma, and C.-H. Hsu, “Dnnoff: Offloading dnn-based intelligent iot applications in mobile edge computing,” *IEEE Transactions on Industrial Informatics*, vol. 18, no. 4, pp. 2820–2829, 2022.

[186] J. Ren, H. Wang, T. Hou, S. Zheng, and C. Tang, “Federated learning-based computation offloading optimization in edge computing-supported internet of things,” *IEEE Access*, vol. 7, pp. 69 194–69 201, 2019.

[187] A. E. Eshratifar and M. Pedram, “Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment,” in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, ser. GLSVLSI ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 111–116.