Lightweight Protocols and Applications for Memory-Based Intrinsic Physically Unclonable Functions Found on Commercial Off-The-Shelf Devices

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von M.Sc. André Schaller aus Zeitz
Tag der Einreichung: 18.10.2017, Tag der Prüfung: 29.11.2017
Darmstadt 2017 — D 17

1. Gutachten: Prof. Dr. Stefan Katzenbeisser
2. Gutachten: Prof. Dr. Jakub Szefer

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Security Engineering Group
Fachbereich Informatik
Computer Science Department

Lightweight Protocols and Applications for Memory-Based Intrinsic Physically Unclonable Functions Found on Commercial Off-The-Shelf Devices

Genehmigte Dissertation von M.Sc. André Schaller aus Zeitz

1. Gutachten: Prof. Dr. Stefan Katzenbeisser
2. Gutachten: Prof. Dr. Jakub Szefer

Tag der Einreichung: 18.10.2017
Tag der Prüfung: 29.11.2017

Darmstadt — D 17

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18.10.2017

(M.Sc. André Schaller)

Zusammenfassung

Derzeit erleben wir die Ära, in der durch die zunehmende Verbreitung von miteinander verbundenen, eingebetteten Geräten sich das sogenannte „Internet of Things“ (IoT) manifestiert. Obwohl derartige Endgeräte oft als „smarte Objekte“ bezeichnet und ihnen somit eine gewisse Intelligenz unterstellt wird, ist aus Perspektive der IT-Sicherheit oftmals das Gegenteil der Fall. Die Einwirkung der Märkte im Bereich kommerzieller IoT-Geräte führt zur Minimierung der Herstellungskosten sowie der Produkteinführungszeiten. Dieser weit verbreitete Trend hat einen direkten, desaströsen Einfluss auf die Sicherheitseigenschaften der Geräte. Eine Großzahl aktuell genutzter und zukünftig produzierter Plattformen implementieren keinerlei oder nur unzureichende Sicherheitsmechanismen. Vor allem das Fehlen sicherer Hardware-Komponenten, unterbinden oftmals die Implementierung sicherer Protokolle und Anwendungen.

Diese Arbeit widmet sich einem grundsätzlichen Lösungsansatz, welcher es erlaubt, kommerzielle, eingebettete Geräte, die durch das Fehlen sicherer Hardware-Komponenten verschiedensten Angriffen ausgesetzt sind, nachträglich absichern kann. Im speziellen, nutzen wir das Konzept der Physikalisch Unkopierbaren Funktionen (PUFs), um Hardware-basierte Sicherheitsanker in Standard-Hardware-Komponenten zu erstellen. Wir setzen dabei auf die Varianz von Static Random-Access Memory (SRAM) und Dynamic Random-Access Memory (DRAM) Modulen, welche während der Herstellungsprozesse induziert werden, um intrinsische, speicherbasierte PUF Instanzen zu extrahieren und darauf aufbauend, sichere und leichtgewichtige Protokolle und Anwendungen zu entwickeln.

In einem ersten Schritt führen wir zu diesem Zwecke eine empirische Evaluation ausgewählter und repräsentativer Gerätetypen bezüglich ihrer PUF-Eigenschaften durch. Als nächstes nutzen wir die Gerätetypen, welche sich durch die Existenz guter PUF-Instanzen für die darauf folgende Entwicklung leichtgewichtiger Sicherheitsanwendungen und -protokolle qualifizieren. Im zweiten Teil der Arbeit stellen wir verschiedene Software-basierte Sicherheitslösungen vor, die speziell für die charakteristischen Eigenschaften von eingebetteten Geräte konzipiert sind. Im speziellen umfassen diese Lösungen eine Secure Boot Architektur sowie einen Ansatz, die Integrität von Firmware durch Bindung an die zugrundeliegende Hardware zu schützen. Weiterhin wird ein leichtgewichtiges Authentifizierungsprotokoll vorgestellt, welches die Charakteristiken eines neuartigen DRAM-basierten PUF-Typs ausnutzt. Als letztes wird ein Protokoll vorgestellt, welches erlaubt, die Sicherheit des Software-Zustandes von entfernten, eingebetteten Geräten sicher zu attestieren.

Abstract

We are currently living in the era in which through the ever-increasing dissemination of interconnected embedded devices, the Internet-of-Things manifests. Although such end-point devices are commonly labeled as “smart gadgets” and hence they suggest to implement some sort of intelligence, from a cyber-security point of view, more then often the opposite holds. The market force in the branch of commercial embedded devices leads to minimizing production costs and time-to-market. This widespread trend has a direct, disastrous impact on the security properties of such devices. The majority of currently used devices or those that will be produced in the future do not implement any or insufficient security mechanisms. Foremost the lack of secure hardware components often mitigates the application of secure protocols and applications.

This work is dedicated to a fundamental solution statement, which allows to retroactively secure commercial off-the-shelf devices, which otherwise are exposed to various attacks due to the lack of secure hardware components. In particular, we leverage the concept of Physically Unclonable Functions (PUFs), to create hardware-based security anchors in standard hardware components. For this purpose, we exploit manufacturing variations in Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory modules to extract intrinsic memory-based PUF instances and building on that, to develop secure and lightweight protocols and applications.

For this purpose, we empirically evaluate selected and representative device types towards their PUF characteristics. In a further step, we use those device types, which qualify due to the existence of desired PUF instances for subsequent development of security applications and protocols. Subsequently, we present various software-based security solutions which are specially tailored towards to the characteristic properties of embedded devices. More precisely, the proposed solutions comprise a secure boot architecture as well as an approach to protect the integrity of the firmware by binding it to the underlying hardware. Furthermore, we present a lightweight authentication protocol which leverages a novel DRAM-based PUF type. Finally, we propose a protocol, which allows to securely verify the software state of remote embedded devices.

Acknowledgements

First and foremost, I am deeply indebted to my supervisor professor Stefan Katzenbeisser. His profound expertise, his supervision and his unrestricted support contributed greatly to the successful results of my research and eventually to the compilation of this doctoral thesis. In fact, his supervision did not just shape the process of my scientific work but also me as a person. Thus, I want to state explicitly that without him the successful writing of this thesis would not have been possible. In addition, I would like to thank professor Jakub Szefer, who enabled me to spent two months researching at the Computer Architecture Laboratory Lab (CAS Lab) at Yale University. This visit allowed me to work on exciting research projects and to exchange inspiring ideas with his group, which eventually resulted in several publications. Moreover, I express my thanks to my colleagues of our research group, the Security Engineering group. Our longtime collaboration did not just open up new vistas and led to innovative solutions but also led to friendships. My special thanks go to my family: my parents Ingrid and Gerhardt Schaller, my sister Katja and my brother-in-law Finley. Their continuous support, trust and positivity provided me with the energy necessary to successfully handle this challenge. Finally, I would like to thank my many friends for their permanent support.

Danksagung

In erster Linie gilt mein Dank meinem Betreuer Professor Stefan Katzenbeisser. Sein profundes Fachwissen, seine Betreuung und seine uneingeschränkte Unterstützung trugen maßgeblich zu den erfolgreichen Resultaten meiner Forschung und schließlich zur Erstellung dieser Doktorarbeit bei. Tatsächlich hatte seine Betreuung nicht nur Einfluss auf den Prozess meiner wissenschaftlichen Arbeit, sondern auch auf mich als Person. Daher möchte ich ausdrücklich festhalten, dass ohne ihn, die erfolgreiche Erstellung dieser Arbeit nicht möglich gewesen wäre. Mein Dank gilt weiterhin Professor Jakub Szefer, welcher mir ermöglichte, für zwei Monate am Computer Architecture Laboratory (CAS Lab) der Yale Universität zu forschen. Durch diesen Besuch konnte ich an spannenden Forschungsthemen arbeiten, trat in einen inspirierenden Austausch mit seiner Forschungsgruppe, woraus letztendlich mehrere Publikationen resultierten. Darüber hinaus möchte ich mich bei meinen Kollegen unserer Forschungsgruppe — der Security Engineering Group — bedanken. Die langjährige Zusammenarbeit mit ihnen eröffnete nicht nur neue fachliche Perspektiven, führte zu innovativen Lösungsansätzen, sondern brachte auch Freundschaften hervor. Mein besonderer Dank gilt meiner Familie: meinen Eltern Ingrid und Gerhardt Schaller, meiner Schwester Katja und meinem Schwager Finley. Ihre kontinuierliche Unterstützung, ihr Vertrauen und Positivität gaben mir die Energie, welche notwendig war, diese Herausforderung erfolgreich zu meistern. Nichtzuletzt bedanke ich mich bei meinen vielen Freunden, für deren ständige Unterstützung.

Denn indem ein Mensch
mit den ihm von Natur gegebenen Gaben
sich zu verwirklichen sucht,
tut er das Höchste und einzig Sinnvolle,
was er kann.

Hermann Hesse

Gewidmet meinem Großvater Horst Künzel.

Contents

List of Figures

List of Tables

Acronyms

AES Advanced Encryption Standard. 39, 66

ASIC Application-Specific Integrated Circuit. 3, 21, 31, 45

COTS Commercial Off-The-Shelf. 2–6, 10, 13, 14, 19, 26, 31–34, 40, 42, 53, 60, 69, 100, 101, 103, 105

CPU Central Processing Unit. 2, 4, 11, 12, 26

CTW Context Tree Weighting. 33, 39, 40, 46

DRAM Dynamic Random-Access Memory. XI, XIII, 2–7, 9, 11–15, 17–21, 23, 26–31, 33–39, 46–51, 53–56, 58–60, 62, 64, 66, 78, 87–92, 105, 106

ECC Error-Correcting Code. 22, 29, 31

EEPROM Electrically Erasable Programmable Read-only Memory. 10–12, 14, 34, 35, 62, 102

FE Fuzzy Extractor. 22, 73, 74, 82

FPGA Field-Programmable Gate Array. 3, 21, 26, 31, 45

HCI Hot Carrier Injection. 18, 19

HMAC Keyed-Hash Message Authentication Code. 96–98, 101–103

IoT Internet-of-Things. 1, 5, 10, 24, 93, 95, 96, 99

JTAG Joint Test Action Group. 12, 61, 95, 101, 102

KDF Key Derivation Function. 96, 103

MCU Microcontroller Unit. XI, XII, 1, 2, 4, 9–12, 14, 19, 23, 25, 31–36, 41, 42, 44–46, 59–65, 69, 95, 97, 99–101, 103

MITM Man In The Middle. 86–88

NVM Non-Volatile Memory. 12, 22, 61–64, 67, 68, 88

OS Operating System. 27, 36

PCB Printed Circuit Board. 4, 35

PoP Package-on-Package. 11, 26, 35, 107

PRNG Pseudo-Random Number Generator. 73, 74, 76, 77, 79, 80, 82–84

PUF Physically Unclonable Function. VIII, IX, XI–XIII, 3–7, 9, 14–60, 62–65, 67, 69, 71, 73, 74, 79–84, 87–92, 101–103, 105–107

RFID Radio-Frequency Identification. 10, 86, 87

ROM Read-Only Memory. XI, 22, 26, 62, 63

RoT Root of Trust. 4, 12, 13, 95, 97, 98, 100–102

SGX Software Guard Extensions. 2, 13, 22

SoC System on Chip. XI, XII, 1, 10, 11, 13, 25, 26, 33–35, 62–65, 92, 95, 98, 100, 101, 107

SRAM Static Random-Access Memory. XI, XIII, 2–7, 9, 11–15, 17–19, 22–26, 31–46, 50, 53, 56, 59, 60, 62–65, 67–69, 71, 73, 74, 81, 82, 84, 98, 101–103, 105

TEE Trusted Execution Environment. 2, 13, 61, 96

TPM Trusted Platform Module. 2, 13, 22, 70, 93, 96, 98, 99

UART Univeral Asynchronous Receiver Transmitter. 25, 36, 40, 82

Chapter 1

Introduction

In the Internet-of-Things (IoT) [LTL16] embedded devices, particularly low-cost and resource-constrained platforms are becoming the fundamental building blocks for many facets of life. Innovation in this space is not only fueled by making devices ever more powerful, but also by a steady stream of even smaller, cheaper, and less energy-consuming "smart things" that enable new features and greater automation at home, in transportation [Sye+12], smart factories [Zue08; Zue10] and cities [Zan+14]. Primarily, miniaturization and cost reduction of Microcontroller Unit (MCU) and System on Chip (SoC) designs have enabled the creation of almost ubiquitous smart devices, from smart thermostats and appliances to smart phones and embedded car entertainment systems. Besides private applications of embedded platforms, they are employed to enhance the degree of industrial automation and in particular, to realize "Industry 4.0" [Jaz14]. Generally in the industrial domain, embedded devices are increasingly used as integral part of more complex Cyber-Physical Systems (CPS) [LS16]. CPS are commonly leveraged in production systems, for the purpose of process monitoring and controlling or to realize responsive quality management [LCK16]. Substituting traditional components in this domain such as programmable logic controllers (PLC) and remote terminal units (RTU), embedded systems are employed to sense and control physical processes as well as communication interfaces for (existing) industrial automation systems to the Internet or an internal network [Jaz14].

Unfortunately, the novelty of this field combined with dominating market forces to minimize cost and time-to-market also has a devastating impact on security. In particular, one major concern is that these devices often lack hardware as well as software implementations of sufficient security mechanisms [Sch14b; VT12]. As a result, the proliferation of such "smart" devices leads to a constant discovery of new security vulnerabilities [Her+14; Cos+14; Def15; Gre15; Fos+15]. Moreover, constraints on the available memory and computational power have an impact on the security of these devices. Means to provide robust identification and authentication of involved devices and mechanisms to securely store long-term cryptographic keys, while minimizing the chances of their illegitimate extraction or access, are particularly demanding tasks for this device class. Besides interconnected platforms, the ability to recognize and establish trust in embedded devices is even becoming relevant for ultra low-cost storage devices, including SD-cards and USB sticks that are exchanged with third parties and hence can be infected or replaced by malicious hardware in order to attack the host [b313; NL14]. Moreover, Bluetooth devices may offer an even larger attack surface since typically employed security mechanisms were shown to be insufficient [Rya13; SW05].

In order to establish trust in these resource-constrained devices, respective security solutions must be based on realistic assumptions concerning the specific hardware capabilities of these devices. Hence, a meaningful approach of designing security protocols and applications towards the peculiarities of low- cost and embedded devices begins with defining requirements of such solutions, taking into account the limitations imposed by the underlying platforms. A first requirement is imposed by the device class considered in this thesis, namely *Commercial Off-The-Shelf (COTS) embedded devices*, which usually do not implement secure hardware such as Trusted Platform Module (TPM), Intel Software Guard Extensions (SGX) or ARM TrustZone for economic reasons. Hence, secure protocols and applications cannot rely on dedicated secure hardware, such as TPM or a Trusted Execution Environment (TEE). Instead, they are usually restricted to *standard hardware components*. Hereafter we denote standard hardware as such components that are produced for general-purpose usage and which are not application specific. Common examples for standard hardware components comprise Static Random-Access Memory (SRAM), Dynamic Random-Access Memory (DRAM), Central Processing Units (CPUs) and Graphics Processing Unit. A second important requirement derives from the very limited computational resources and memory-related capabilities of such devices. In particular, a security solution only qualifies if it exhibits a small memory footprint and if it relies on building blocks that are computationally efficient. Cryptographic protocols that are specifically optimized towards such constrained devices, are commonly denoted as *lightweight security protocols* [EK07]. Hence, the adoption of classical yet more heavyweight approaches, such as most asymmetric cryptographic systems, is not a viable option for many MCUs-based device types (cf. Chapter 2.2). In particular, for ultra low-cost 8 bit microcontrollers, that are found in smart cards or RFID chips, efficient implementation of traditional cryptographic solutions is a non-trivial task. From the perspective of the memory domain (8 bit MCUs are usually limited to 256 B to 1 kB RAM), implementations of public-key cryptographic systems are indeed feasible, especially when leveraging ECC that uses smaller key sizes compared to RSA, to achieve equivalent security levels [Bro09]. However, the time required for decryption, encryption, signing or verifying a signature can be tremendous. As shown in [Pie00], RSA suffers from computing expensive decryption, which can take up to several hundreds of seconds using a 2048 bit key on an 8 MHz platform. Leveraging ECC, the decryption process was shown to be inefficient on a similar platform, requiring up to 8 s using a 193 bit key, with both key sizes of RSA and ECC providing comparable security levels. Further approaches towards efficient implementations of public-key cryptographic systems and signature schemes show similar runtime results, which are in the same order of magnitude [PLP06; LGK10]. In many applications these performance characteristics are not acceptable. Moreover, such devices do not allow for hardware customization after their acquisition “off- the-shelf” and obviously not after their deployment to their field of operation. Thus, a third vital requirement to any security solution applicable to such devices is to be a purely *software-based* solution, i.e., to not require any hardware modifications. Instead, solutions must build on existing hardware capabilities a well as features provided by the firmware drivers and libraries. In order to establish better understanding of the hardware capabilities of the targeted device class, we provide a detailed description of the device class considered in this thesis in Chapter 2.2. At last, a fourth requirement for successfully addressing the challenge of establishing trust in commercially available embedded devices is to consider the major phases of utilization of such devices firmware[1] and its interaction with remote entities in its entirety. In particular, major distinct execution phases of firmware on embedded devices comprise i) the boot phase conducted by the boot loader, ii) execution phase of the firmware application and iii) authentication towards external interconnected entities. Furthermore, especially in the case of low-end embedded devices, the possibility to attest the level of trust and to re-establish trust in case of being compromised by malware (i.e., “device recoverability”) is becoming an increasingly vital requirement. We argue that only a *holistic approach* that considers all these aspects

[1] In this work, we will use the terms software and firmware interchangeably.

that emerge during the interaction with COTS embedded platforms suffices in order to establish trust in said platforms.

With those identified requirements at hand, providing strong security guarantees becomes a challenging task. In this thesis we propose to address these challenges by leveraging the concept of *memory-based intrinsic Physically Unclonable Functions (PUFs)* to provide a hardware-enforced security anchor on low-cost devices, which can be established efficiently and by software-only means. PUFs exploit device-unique physical characteristics of the underlying hardware, that can be used to derive “fingerprints” of hardware components. Based on these device-unique fingerprints, cryptographic keys can be extracted that allow for the subsequent employment of lightweight security protocols and applications on the targeted devices.

Although PUFs have been made a subject of discussion in previous works in the literature, up to now the majority of research has considered PUFs to be a security building block, which has to be added explicitly during the design phase of the chip (i.e., either as part of an Application-Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA)) before it can be utilized, in turn requiring a priori design decisions. As a result, most low-cost devices, which are usually optimized towards cost-efficient manufacturing and hence do not allow for modifications of the production process, have not been addressed in existing research.

However, many commercially available devices already contain standard components that can be leveraged for PUF use. In particular, some memory modules already present in COTS devices – specifically SRAM and DRAM – exhibit good PUF behavior, even though such modules were not designed for the purpose of using them as PUFs in the first place. In the first part of this thesis, we investigate whether it is possible to find and utilize existing PUF instances in commercially available low-cost embedded devices. Having found usable PUF instances, the second part of this thesis is dedicated to the development of lightweight security protocols that leverage those PUF instances in order to eventually establish trust in those otherwise unprotected low-cost devices. In particular, we present the following PUF-enabled applications and protocols that provide security for the phases of interaction with embedded low-end devices:

1. Secure boot of the device firmware application.
   The presented solution provides a software-only mechanism to guarantee *boot time integrity* of firmware application. For this purpose we leverage the intrinsic SRAM PUF instance, in order to derive a device-unique key, which is subsequently used to decrypt the firmware application by an immutable boot loader. Hence, this solution effectively protects against firmware replacement or extraction on low-end embedded devices.

2. Integrity protection during execution.
   Having secured the loading of the firmware application, a next step is to protect the *execution time integrity* of the application, i.e. to prevent against analysis and modification at runtime. TO this end, we combine the technique of self-checksumming codes with SRAM-based PUFs to protect against modification of the firmware execution and binding of the same to the underlying platform.

3. Lightweight mutual device authentication.
   We present a very lightweight protocol that allows for *mutual authentication* of heavily resource-constrained devices with an external verifier. While we leverage a novel DRAM-based PUF construction, we avoid computational expensive error-correction, which is traditionally required to derive a robust key from PUF instances and hence would otherwise render PUF usage impractical on very resource-constrained devices. Instead, we leverage

the characteristics of the underlying DRAM-based PUF to realize a more lightweight and robust mutual authentication scheme.

4. Remote attestation and device recovery.
   We propose a novel attestation scheme entitled “Boot Attestation”, which is the first *remote attestation protocol* that is optimized towards highly resource-constrained off-the-shelf devices. It supports a variety of standard hardware components, intrinsic PUFs amongst others, which are used as a Root of Trust (RoT). From this RoT and based on a symmetric attestation key, software integrity of the individual firmware stages is measured and committed immediately, resulting in a so-called “Chain of Trust”. The scheme allows for device recovery after runtime compromise and can be extended to support various read-world use cases, such as third party verification or key provisioning.

1.1. Research Goal and Research Question

Addressing the class of commercial embedded devices that lack dedicated secure hardware mechanisms, the research goal of this work is to establish trust in such platforms by leveraging existing on-board memory components as PUF instances, with particular focus on SRAM and DRAM modules. For this purpose, this thesis tries to answer the following research question:

Do standard memory components, implemented on commercial off-the-shelf microcontrollers, show physical variations that exhibit PUF-like behavior, such that these components can be exploited as hardware-based security anchors in lightweight security protocols?

Hence, the focus of this thesis lies on the physical variations of said memory components found on COTS embedded devices, which will be analyzed with regards to their PUF characteristics and which subsequently serve as a basis for constitutive security protocols and applications. By focusing on the physical layer, i.e., considering standard hardware components, which are intrinsically part of commercial low-cost platforms as security primitives, we achieve manifold desired properties:

1. By focusing on the physical level of standard hardware components, i.e., SRAM and DRAM modules, which can be found on the majority of COTS devices, we achieve a high degree of independence from vendor-specific implementation decisions, including individual layouts of the Printed Circuit Board (PCB), on- or off-chip organization or interaction of functional elements of the MCU that resides at the core of all the considered device classes (cf. Chapter 2.2). Functional elements typically comprise the CPU, memory components, timers and external peripheral components.

2. By relying on standard hardware components, we allow for developing secure protocols and applications that help to maintain trust in COTS embedded devices, despite the absence of secure hardware components, which are otherwise required in traditional security solutions [Tru11].

3. By carefully designing secure protocols and applications solutions to require software-only modifications, any changes to the hardware are avoided entirely. Hence, implementation of PUF-based security solutions is simplified tremendously, immediately providing support of a broad range of COTS devices.

4. Lastly, we allow for retro-fitting hardware-based security anchors to legacy devices, which are already in-field, in order to deploy supplementary security features.

1.2. Methodology

In order to answer the research question and to efficiently realize the research goal, the first part of this work deals with the identification, extraction and evaluation of PUF instances extracted from standard hardware found on COTS embedded devices. The second part of this thesis presents secure software solutions, i.e. secure boot, code integrity protection, mutual authentication and remote attestation, that are based on the PUF instances previously identified. Moreover, the underlying device types are leveraged as evaluation platforms to prove the feasibility and efficiency of the implementations. From this methodical approach a gradual process results as follows:

1. In a first step, commercial embedded platforms are identified, which have a large scope of application among typical use cases of IoT and thus are qualified representatives for the targeted device class.
2. After identifying wide-spread embedded platforms, means to extract PUF instances from each device's SRAM and DRAM modules are identified. This step involves physically measuring the targeted hardware components in order to extract a significant number of PUF measurements. The extraction process must be software-only, i.e., without requiring any hardware modifications that are usually infeasible on COTS platforms.
3. The measurements obtained in the previous step serve as the data set for subsequent evaluation of the identified PUF instances towards various PUF characteristics. Evaluation is done in order to be able to quantify PUF characteristics that have an influence on the security of the overlying protocols.
4. Finally, based on those PUF instances that yield positive evaluation results with respect to their PUF characteristics, security protocols and applications were developed in order to show how to establish trust in these devices.

1.3. Thesis Outline

Given the research methodology introduced in this chapter, the remainder of this thesis is structured as follows.

Chapter 2 introduces fundamental notations used throughout the remainder of this work. Furthermore, the targeted device class is discussed, focusing on common memory technologies found on such devices, SRAM and DRAM in particular.

The concept of Physically Unclonable Functions is introduced and defined in Chapter 3. Moreover, central properties that are used to assess and quantify PUF characteristics are discussed. Furthermore, basic principles of the three PUF types built from SRAM and DRAM modules are explained.

In Chapter 4 we identify a set of device types which are representatives of COTS embedded devices and which serve as evaluation and implementation platforms throughout this thesis. Furthermore, a common software-only approach to extract memory-based intrinsic PUF instance from these devices is presented. Finally, this chapter details the evaluation process of the extracted PUF instances and discusses its results.

After showing the existence of favorable memory-based intrinsic PUF instances on COTS platforms, Chapter 5 presents a first application which builds on these PUF instances. In particular, a lightweight secure boot solution is proposed, that protects against firmware extraction on embedded devices by exploiting SRAM-based PUFs.

Subsequently, Chapter 6 extends the secure boot application by introducing a PUF-based solution that focuses on the protection of the firmware integrity. Again, SRAM-based PUF instances are used as a hardware-based trust anchor, which are combined with the technique of self-checksumming code in order to protect from firmware modifications.

Chapter 7 is dedicated to the task of authenticating a given low-cost device towards a remote server. As there are numerous existing approaches to device authentication for SRAM-based PUFs, this section presents the first lightweight authentication protocol that uses the decay-based DRAM PUF, which was previously introduced in this work.

In Chapter 8 a novel and ultra lightweight remote attestation scheme is presented. As the scheme relies on memory-based intrinsic PUFs, amongst other hardware primitives, it allows for adoption to a wide-spread spectrum of COTS and legacy devices.

Finally, this thesis is concluded in Chapter 9, summarizing the contributions and discussing potential future work.

Chapter 2

Preliminaries

In this chapter we discuss important concepts, which will be used frequently throughout this thesis. On the one hand, these concepts will serve as a basis for the evaluation of Physically Unclonable Function (PUF) instances and are used to infer properties of the PUF-enabled protocols. On the other hand, concepts introduced in this chapter illustrate the technical properties of the involved device types and the memory components considered in this work. Since knowledge of these underlying concepts greatly contributes to the understanding of the remainder of this thesis, they are discussed in detail.

In particular, we will first introduce basic mathematical and information- theoretic notations, which will be used extensively during evaluation of the PUF instances in Chapter 4. Next, we model the targeted device class by identifying common hardware capabilities and general properties. Lastly, we detail the physical structure of the investigated memory components, namely Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory (DRAM), in order to provide an understanding of the involved physical effects that result in PUF behavior.

2.1. Notations

Throughout this thesis the following notation and definitions are used frequently, mainly to model the characteristics of a PUF or to infer information-theoretical statements about the privacy or security of PUF-enabled applications.

Variables, Distributions and Sets

Random variables are written in capital letters, such as X. A particular instance of the variable is written in lowercase, i.e., x. The set of possible instances of X and sets in general are denoted with a calligraphic symbol $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$. The probability function of a random, discrete variable X is defined as

$$p(x) = Pr(X = x). \tag{2.1}$$

We denote the process of uniformly randomly sampling from a distribution X as $x \overset{\$}{\leftarrow} \mathcal{X}$.

2.1.1. Functions and Mappings

We denote a function f that maps elements from X to values of Y as $f : X \mapsto Y$. In cases where we expect X to consist of bitstrings of length m and elements of Y to be bitstrings of length n, we write $f : X^m \mapsto Y^n$.

2.1.2. Distance and Similarity Metrics

Hamming weight

The Hamming weight **HW** is a metric that defines the number of non-zero symbols in a sequence of symbols $a = \{a_1, a_2, \ldots, a_n\}$. In the classical case and in the context of this thesis, a is assumed to be a bitstring, such that **HW** depicts the number of non-zero elements of a:

$$\mathbf{HW}(a) \overset{def}{=} |\{i \in \{1, 2, \ldots, n\} | a_i \neq 0|. \tag{2.2}$$

Hamming Distance

The Hamming distance **HD** between two symbol sequences $a = \{a_1, a_2, \ldots, a_n\}$ and $b = \{b_1, b_2, \ldots, b_n\}$ of same length n and generated from the alphabet Σ, denotes the number of positions at which both symbol sequences differ:

$$\mathbf{HD}(a, b) \overset{def}{=} |\{i \in \{1, 2, \ldots, n\} | a_i \neq b_i|. \tag{2.3}$$

Jaccard index

The Jaccard index $\mathbf{J}(\cdot, \cdot)$ is a metric that reflects the similarity of two sample sets $\mathcal{A}$ and $\mathcal{B}$. Here, the sample sets can be of unequal size. It is computed as:

$$\mathbf{J}(\mathcal{A}, \mathcal{B}) \overset{def}{=} \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}, \text{ with } 0 \leq J(\mathcal{A}, \mathcal{B}) \leq 1. \tag{2.4}$$

The Jaccard index is '1' if the sets $\mathcal{A}$ and $\mathcal{B}$ are identical. Likewise, the index is '0' if both sets share no common elements.

2.1.3. Entropy

In order to make statements about the information content, i.e., entropy of a PUF, we introduce the following notations. The notation 'log' refers to the base-2 logarithm.

Shannon entropy

The Shannon entropy [Sha48] $\mathbf{H}(X)$ of a discrete, random variable X is denoted as

$$\mathbf{H}(X) = -\sum_{x \in X} p(x) \log p(x). \tag{2.5}$$

Binary entropy

Suppose that X is a binary random variable, i.e.,

$$X = \begin{cases} Pr(X=1) = p \\ Pr(X=0) = 1-p. \end{cases} \quad (2.6)$$

The binary entropy function $\mathbf{H_b}(X)$ is written as

$$\mathbf{H_b}(p) \overset{def}{=} -p\log p - (1-p)\log(1-p). \quad (2.7)$$

Min-entropy

Furthermore, for a binary source X, we define the min-entropy [Rén+61] $\mathbf{H}_\infty(X)$ as

$$\mathbf{H}_\infty(X) = -\log\big(\max((p(0), p(1))\big), \quad (2.8)$$

where $p(0)$ and $p(1)$ are the probabilities of $P(X=0)$ and $P(X=1)$, respectively. Note that the min-entropy denotes the greatest lower bound on the entropy for X.

Mutual information

The mutual information [CT12] $\mathbf{I}(X,X')$ between two discrete variables X and Y is denoted by

$$\mathbf{I}(X;Y) = \sum_{y\in Y}\sum_{x\in X} p(x,y)\log\left(\frac{p(x,y)}{p(x)p(y)}\right). \quad (2.9)$$

Informally, it defines the amount of information that can be obtained about a variable X through a second variable Y. Here, $p(x,y)$ denotes the joint probability function of X and Y.

2.2. Common Capabilities of Low-Cost Devices

2.2.1. Hardware

In the following section, we will define the classes of devices considered throughout this thesis, as selected device types of this class will serve as a basis for evaluation of PUF instances. Furthermore, they will be used as proof-of- concept platforms for the implementation of PUF-enabled security applications. For this purpose we first outline the capabilities of respective device types, regarding their computational and memory faculties and subsequently identify hardware components that are common to devices of this class. At last, we detail the principles of SRAM and DRAM, as two widely-used memory technologies on Microcontroller Unit (MCU)-based platforms.

Device Class	Processing Power	Volatile Memory (SRAM)	Non-Volatile Memory (Flash / EEPROM)	Usage Scenario
Class-0	$\leq$ 32 MHz	$\leq$ 6 kB	$\leq$ 32 kB	home automation, sensor networks, smart objects
Class-1	32 MHz to 120 MHz	8 kB to 32 kB	32 kB to 256 kB	home automation, industrial environments, networking appliances
Class-2	$\gg$32 MHz	32 kB to 512 kB	$\gg$1 MB	smart car infotainment, wearable devices, industrial environments

Table 2.1.: The three classes of embedded devices evaluated in this thesis.

We consider the space of so-called *Commercial Off-The-Shelf (COTS) embedded devices*, which are increasingly deployed as important building blocks of the Internet-of-Things (IoT).

The predicate *COTS* mainly refers to the property that such devices are manufactured in mass production to yield large quantities. Hence, they are commercially available and are comparably cheap to obtain. Furthermore, they tend to be adaptable to various applications that share similar performance requirements.[1] Their cost-efficiency as well as their versatility commonly lead to their high dissemination.

The term *embedded devices* lacks a rigorous definition and incorporates various device classes which observe different hardware facilities. The main common characteristic of the device types considered as embedded devices is the much higher restriction of their (hardware) capabilities, compared to desktop PCs, laptops, servers and alike. Hence, the embedded attribute suggests that these devices tend to be single-purpose devices with often the minimum memory required to meet their intended application, in order to minimize manufacturing cost and power consumption. While these restrictions manifest in constraints on processing resources, available memory, electrical energy consumption and interconnectivity, the main characteristic of embedded devices from a security point of view lies in the unavailability of dedicated secure hardware. However, there are classes of embedded devices that have quite diverse characteristics, i.e., varying processing power (i.e., MCUs with up to 32 MHz vs. System on Chips (SoCs) running at 1 GHz) or which implement different architectures (4 bit vs. 32 bit).

In order to introduce some structure to this rather undifferentiated pool of embedded devices and to provide a way of relating the evaluated device types to a set of common hardware capabilities, we introduce three device classes. For this purpose we follow the categorization of [BEK14] which introduced the notion of *constrained* devices. The spectrum of embedded devices extends from the class of ultra low-cost MCU-based class-0 devices (e.g., used in Radio-Frequency Identification (RFID) tokens) through to conventional MCUs (often deployed in industrial environments) that belong to class-1, to class-2 devices that resemble sophisticated, complex SoC platforms (i.e., as part of in-car infotainment systems). Table 2.1 provides an overview of the characteristics and hardware capabilities associated with each device class. Given the vast variety of device types that observe numerous configurations of processing power, volatile memory, non-volatile memory (i.e., flash end Electrically Erasable Programmable Read-only Memory (EEPROM)), the presented numbers do not act as sharp class boundaries but are rather used as orientation values indicating the capabilities of the corresponding hardware characteristic.

[1] In contrast to high-priced hardware, which is made for general purpose usage, most embedded devices are restricted to a subset of applications, due to their limited hardware capabilities.

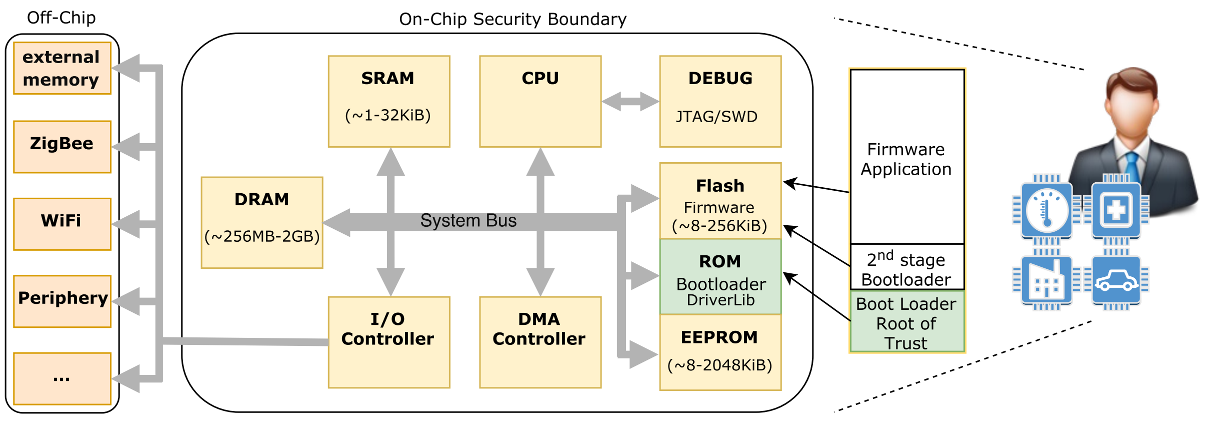


Figure 2.1.: Typical hardware configuration of an embedded MCU.

Although SoC-based device types are more complex in their structure and have better hardware capabilities compared to class-0 devices, at the core of all embedded devices resides an MCU, featuring the Central Processing Unit (CPU), memory components, I/O controllers and a debugging interface on a single integrated circuit. Ultimately, the most basic device types belonging to class-0 entirely consist of a single MCU. In contrast, class-2 device types are usually made up of several MCUs. The MCU is highly-integrated, implementing SRAM, flash and sometimes EEPROM as part of a single die package, i.e., such components are usually "on-chip" (cf. Figure 2.1). Although DRAM is currently implemented as an external component, i.e. "off-chip", we foresee that due to the fast maturation of embedded DRAM technology, future embedded devices will also feature DRAM as an on-chip component (cf. Section 2.3). The high level of integration of key components within a single chip creates an effective protection against attempts to physically probe, modify or entirely replace such components. Compared to a system which implements key components as isolated packaged modules (i.e., a laptop or server), an invasive attacker who wants to tamper with on-chip components of an MCU will be required to put in much more effort in terms of time and hardware equipment. One of the attacker's challenges is to de-package the core chip, keeping the highly interconnected components functional, in order to get access to the components in the first place. This task is becoming even more challenging if state of the art circuit packaging technologies such as Package-on-Package (PoP) are used [Yos+06]. In this case, functional components such as the CPU, SRAM and other on-chip components are vertically organized in a stacked fashion. Hence, the underlying design principle of MCUs and SoCs of high integration establishes an innate security boundary against basic physical adversaries.

The majority of MCUs follow the Reduced Instruction Set Computer (RISC) design principle that allows for less cycles per instruction by using fewer but more general-purpose instructions instead of a higher number of more complex instructions as in the Complex Instruction Set Computer (CISC) architecture. [2] More sophisticated class-2 SoC platforms, such as the OMAP4460 (cf. Chapter 4.2), even implement multi-core architectures. However, even the simplest class-0 MCUs implement some form of parallelism as most microcontrollers follow a pipelined architecture.

[2] Note however, that due the steadily increasing complexity of MCUs, a strict discrimination between both architectures is not always possible. For example, ARM processors, being RISC processors, allow for different instruction sizes by using the ARM Thumb mode, which is a feature of CISC processors.

2.2.2. Firmware

On the software side, MCUs are often programmed bare-metal, with the Software Development Kit (SDK) building necessary drivers and libraries into a single application binary (firmware image). The term firmware usually refers to a fixed piece of software, usually not intended to be changed by the user [AGS12]. The firmware image consists of a *boot loader* part and the application itself, denoted as *firmware application*.

The boot loader part initializes basic hardware components, i.e., it configures the clock, initializes the main memory and sets up key peripherals. Class-0 and class-1 devices usually rely on a very compact single-staged boot loader that (after hardware initialization) merely sets up the stack and calls the firmware application, which usually consists of a `main` function and a loop that continuously executes the application logic at its core. In contrast, class-2 device types often implement a much more complex multi-staged boot loader. Here, the first stage boot loader is small enough to reside in the on- chip SRAM and, amongst others, sets up the CPU, configures the main memory subsystem, initializes and synchronizes various timers and finally sets up external peripherals before invoking the second-stage boot loader. This part is too big to fit into SRAM in its entirety and hence is executed from external DRAM. It is in charge of configuring higher-level functionalities of the embedded device and configuration of the residual peripherals (USB subsystem, networking etc.). One of the most popular multi-staged open source boot loaders is u-boot [Eng02], which is also the standard boot loader implementation of some of the evaluated class-2 device types, including the PandaBoard and the Intel Galileo (cf. Section 4.2). U-boot consists of a first- and a second-stage boot loader. Whilst the first stage initializes core functionalities, the second stage boot loader configures DRAM and external components and eventually executes the application part of the firmware.

The application part implements the main functionality and algorithms of the firmware. The firmware image is stored in Non-Volatile Memory (NVM), usually flash memory or EEPROM. Conventionally, there are two distinct sections of NVM that are dedicated to either the boot loader or the application part of the firmware. While the memory range dedicated to the boot loader is very limited in terms of memory (usually not more then 6 kB to 10 kB on class-1 device types), the entire remaining NVM address space can be used to store the firmware application. On most device types both sections can be configured individually regarding their memory access controls. This is because the boot loader is usually regarded as the Root of Trust (RoT), i.e., the trust anchor in order to boot strap the platform from an integer initial state. Accordingly, the boot loader is usually “locked-down” once it is written to the respective memory region by disabling write-access to it. The firmware application is therefore typically initialized by such an immutable firmware boot loader, which additionally reduces the risk of permanently disabling a device (“bricking”). In order to allow for modification of the boot loader (usually after exceptional events, such as device recovery), most platforms allow for programming via low- level interfaces such as Joint Test Action Group (JTAG). Hence, many device types provide customization of these early stage(s) of boot.

Depending on the underlying platform and application scenario, there are different approaches regarding the firmware’s architecture. Especially in scenarios that focus on low-power (i.e., sensor networks or home automation) or low-cost (i.e., wearables or smart gadgets) mostly MCU-based class-0 device types are employed. Such devices traditionally implement a single-threaded firmware application that continuously executes its functionality within a loop. Systems that operate in time-critical environments (mostly class-1 device types, i.e., as part of industrial control systems or avionics) often implement more complex kernel-based Real-Time

Operating System (RTOS), which allow multitasking by means of event-driven scheduling of multiple tasks [Edi16]. Class-2 SoC-based device types are able to implement an entire rich operating system.

2.2.3. Security Issues

While embedded COTS devices are significantly restricted regarding their hardware facilities, which manifest itself in constraints on processing resources, available memory, electrical energy consumption and interconnectivity, the main characteristic of embedded devices from a security point of view lies in the unavailability of dedicated secure hardware. More powerful computing platforms, such as up-to-date tablets, laptops, desktop PCs and servers traditionally rely on Trusted Platform Modules (TPMs) [Tru11] in order to establish a hardware-based RoT and to provide a foundation for secure protocols and applications. TPM is based on a trust anchor in hardware, the TPM-chip, and provides a set of cryptographic functionalities and protocols. Alternatively, current Intel processors implement the company's own approach towards a Trusted Execution Environment (TEE), called Intel Software Guard Extensions (SGX) [Int16b]. Leveraging an extended instruction set, SGX implements so-called enclaves that act as secure "compartments" used to execute applications in a secure manner. SGX protects the integrity and confidentiality of code and related data run within the enclave from a potentially malicious operating system and further. For this purpose SGX allows to remotely attest the state of an enclave, supports key provisioning and sealing of data communicated with external parties. However, currently only the most powerful processor types implement Intel SGX, omitting Intel's embedded product line. Furthermore, as discussed by Costan and Devadas [CD16], Intel's licensing model, which is integrated in the SGX architecture by means of the Launch enclave and associated patents, suggests that the SGX technology can only be used by a party which maintains some sort of business relationship with Intel. Furthermore, a series of vulnerabilities that can lead to key compromises were discovered recently [Sch+17b]. In the mobile world, the prevailing approach of providing a TEE world is ARM's TrustZone technology [ARM17]. TrustZone leverages an extended bit line in order to create and distinguish between a normal and a secure "world". Using hardware resources, TrustZone provides individual sets of memory, peripheral components and software states for applications running in both worlds. Secure context switching guarantees that no data from the secure world is leaked to the normal world's processes. However, currently only ARM's non-lightweight application processors (ARM Cortex) support TrustZone technology.[3]

Hence, available embedded devices fail to implement dedicated hardware-based RoTs. Furthermore, already deployed devices cannot be retro-fitted with possible future security mechanisms once they are available to the market. Given the high dissemination of such devices and the market force of minimizing costs, their replacement by more secure platforms is highly unlikely. In order to fill this security gap, we propose to use standard hardware components, existing in virtually all computing platforms to realize a hardware-based RoT. In particular, we leverage the manufacturing variabilities of SRAM and DRAM components. Next, we present the principles of both technologies.

[3] ARM recently announced support for a future generation of low-cost architectures, the Cortex-M23 and M33

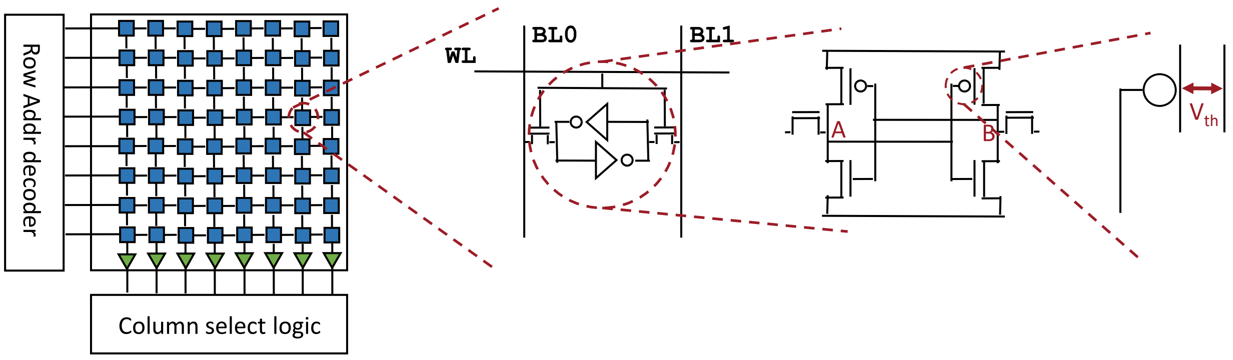


Figure 2.2.: An SRAM array (a) that consists of SRAM cells that form two cross-coupled inverters (b). The structure of a 6T-SRAM cell in detail (c) and the threshold voltage V_{th} of the corresponding transistor (d).

2.3. Memory Technologies Found on Low-Cost Devices

Common on-die memory components that can be found on the majority of low-cost MCUs comprise local instruction and data caches, SRAM[4], flash memory and EEPROM. Recently, embedded on-chip DRAM (eDRAM) technology has become part of newer embedded devices generations. Due to its promising technology and fast improvement [Bar+10; Fre+15; Iye+05; Kal+10; Pei+14; PTT17], we anticipate that DRAM will most likely become a standard on-die element of future low-cost MCUs generations.

Below we will provide background information on the physical structure of SRAM and DRAM technologies, which is necessary to understand the functionality of the PUF types extracted from both technologies, as introduced in Chapter 3. Although common memory components of low-cost COTS MCUs also comprise flash memory and EEPROM – both which have been shown to exhibit PUF-like behavior [Pra+11; Roa+15; Wan+12] – the focus of this work lies on SRAM and DRAM modules.

2.3.1. Static Random Access Memory

Static Random-Access Memory (SRAM) is a standard volatile memory technology that stores information for as long as operational voltage is applied. Although SRAM can be constructed in different ways, the prevailing construction uses six transistors in order to construct a single SRAM cell (6T-SRAM-cell), based on Complementary Metal–Oxide–Semiconductor (CMOS) technology. In this design, four transistors are used to store a single bit, whereas two additional transistors are used to control access to the SRAM cell for reading or writing purposes.

The four storage transistors form two cross-couples inverters (A and B in Figure 2.2), that reinforce each other, effectively creating a positive feedback loop. Once the threshold voltage V_{th} is attained, the respective transistor starts to conduct. Due to manufacturing variations, usually there exists a mismatch between the threshold voltages of both cross-coupled inverters. The inverter with the smaller V_{th} conducts first and initiates the feedback loop. There exist two stable states, used to store a logic zero or one. A third metastable state exists only for a very short

[4] On many embedded platforms, CPU caches are implemented using SRAM technology.

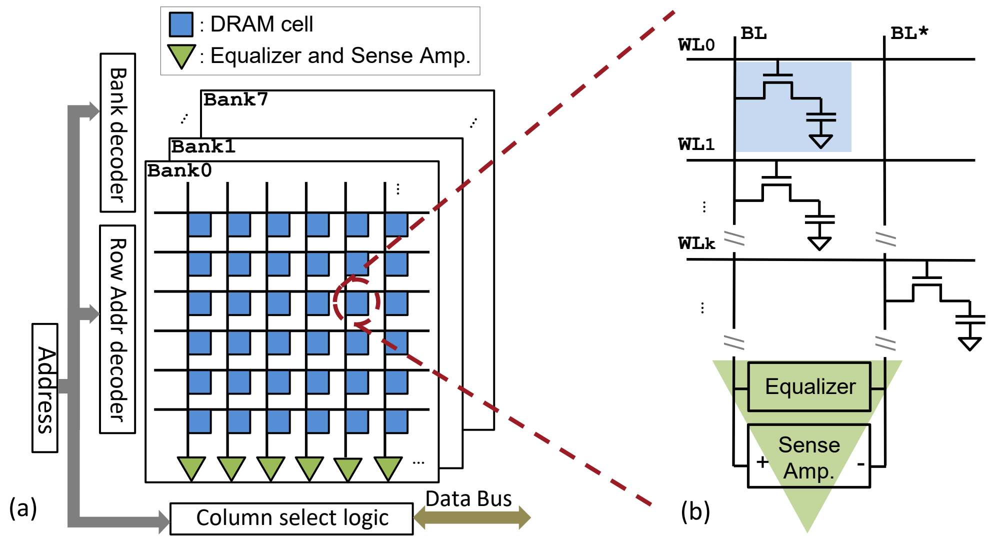


Figure 2.3.: Schematic of DRAM cells (a). DRAM cell organization (b).

time, after supply voltage is applied (i.e., at start-up of the SRAM module). Minuscule deviations from the metastable state are amplified by the feedback loop, pulling the cell to one of the stable states, a process also referred to as “ramp-up”. A schematic description of a 6T-SRAM cell is given in Figure 2.2.

2.3.2. Dynamic Random Access Memory

A single DRAM cell stores a bit charge in a capacitor that can be accessed through a transistor. DRAM cells are grouped into cell arrays, where each row of the array is connected to a horizontal word-line and DRAM cells in the same column are connected to the same bit-line. All bit-lines are coupled to equalizers and sense-amplifiers that amplify voltages on bit-lines to such level that can be interpreted as logical zeros or ones. The structures of a DRAM cell array and a single DRAM cell are depicted in Figure 2.3.

DRAM cells are accessed per row, i.e., the process of accessing a single DRAM cell involves fetching all the neighboring cells that constitute a single DRAM row. In order to access a row, all the bit-lines that are connected to the respective row are pre-charged to half the supply voltage $V_{DD}/2$. Subsequently, the connected word-line is enabled, activating every transistor in that row. This process allows charges form the capacitors to flow to their associated bit-lines. Depending on the charge that was stored on the capacitor, the sense amplifier then drives the bit-line to either V_{DD} or 0 V. Each sense amplifier is usually shared by two bit-lines [Kee08], of which only one can be accessed at the same time. This structure makes the two bit-lines complementary, which results in two kinds of cells: *true-cells* and *anti-cells*. True-cells store the value ‘1’ as V_{DD} and ‘0’ as 0 V on the capacitor, while anti-cells store the value ‘0’ as V_{DD} and ‘1’ as 0 V. A schematic overview of a DRAM array and a single DRAM cell is given in Figure 2.3.

DRAM cells require a periodic refresh of the stored charges, as otherwise the capacitors lose their charge over time, which is referred to as DRAM cell *decay* or *leakage*. The hardware memory controller takes care of periodic refresh, whose interval is defined by the vendor, and is usually 32 ms or 64 ms. We will exploit this decay behavior later, in order to realize the decay-based DRAM PUF in Chapter 3.3.1.

Chapter 3

Physically Unclonable Functions found in Commodity Devices

Published Content

- André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Intrinsic Rowhammer PUFs: Leveraging the Rowhammer Effect for Improved Security.* In: *IEEE International Symposium on Hardware Oriented Security and Trust.* 2017, pp. 1–7.
- Wenjie Xiong, André Schaller, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Run-time Accessible DRAM PUFs in Commodity Devices.* In: *International Conference on Cryptographic Hardware and Embedded Systems.* 2016, pp. 432–453.
- André Schaller, Tolga Arul, Vincent van der Leest, and Stefan Katzenbeisser. *Lightweight Anti-counterfeiting Solution for Low-End Commodity Hardware Using Inherent PUFs.* In: *Trust and Trustworthy Computing.* 2014, pp. 83–100.

In the following chapter, we introduce the notion of Physically Unclonable Functions. The main challenge here is to find a middle ground between a sufficiently precise formal definition and the ability to exhaustively cover the large variations among different PUFs types. In particular, the notion should be accurate enough to cover important characteristics of a given PUF type, which affect privacy or security aspects of constitutive applications and protocols. At the same time, numerous PUF types have been proposed in the literature, which are based on diverse physical phenomena and hence can only be modeled individually if the model is required to cover all involved physical aspects. Thus, the task of defining an accurate, yet all-embracing definition of a PUF is not trivial [Roe12]. Although different existing works tried to approach this task [Arm+11; Arm+16], a sound, unified formal description of PUFs has yet to be found [Mae10; Gan+17].

From an informal perspective, a Physically Unclonable Function (PUF) can be described as a complex physical structure that can be queried by a stimulus c in order to produce a response r. The response r corresponds to a noisy PUF measurement X. The PUF measurement X depends on the challenge c as well as on the micro- or nanoscale physical structure of the underlying object embedding the PUF. Due to minuscule manufacturing variations during the production process, the structure of the physical object containing the PUF is unique to each of its instances. Moreover, as the differences in the physical structure of different PUFs are at a microscopic level, which is supposed to be beyond the influence of the current manufacturing technology, it is assumed that the PUF is unclonable such that it cannot be reproduced, not even by the manufacturer. The challenge-response behavior of the physical system is complex enough such that the response to a randomly selected challenge cannot be predicted.

Below, we give a more detailed definition of PUFs and discuss their properties and resulting areas of applications.

3.1. Physically Unclonable Functions (PUFs)

In contrast to existing approaches in the literature that provide an exhaustive formal definition, this work initially gives an intuitive description of the underlying concept of PUFs and gradually models a PUF by discussing its main properties. Hence, the definition of PUFs given in this chapter is based on their physical properties, with a strong focus on those characteristics that emerge from SRAM and DRAM components. The proposed model of a PUF captures those characteristics that are, in general, common to memory-based PUFs. In this way,s we allow for uniform evaluation of PUF characteristics obtained from SRAM- and DRAM-based PUF types. In turn, this approach allows for making statements about the security and privacy aspects of the protocols and applications that are enabled by both PUF types and which are presented in the latter part of this thesis.

A colloquial definition of the concept of PUFs is given by Maes [Roe12], where the authors state that “a PUF is an object’s fingerprint”. In order to introduce a higher level of precision, we modify this definition to be: *A PUF is a type of function, which is entangled with physical characteristics of a containing object.* Just as traditional, mathematical functions, a PUF f_{PUF} maps input elements (i.e., *challenges*) $\{c_1, c_2, \ldots, c_k\}$ of a challenge space $c_i \in \mathcal{C}$ to output elements (i.e., *responses*) $\{r_1, r_2, \ldots, r_l\}$ of a response space $r_j \in \mathcal{R}$:

$$f_{\mathbf{PUF}} : \mathcal{C}^k \mapsto \mathcal{R}^l. \tag{3.1}$$

We introduce the following simplified notion:

$$r \leftarrow \mathbf{PUF}(c) \ \text{ or } \ \mathbf{PUF}(c) = r. \tag{3.2}$$

3.1.1. Properties

Given the large number of diverse PUF construction that leverage very different physical phenomena, it is a challenging task to find a common subset of properties that allow for a definition whether a given construction or hardware component implements a PUF instance at which level of quality. Thus, in the literature different properties were deemed as essential for a PUF, resulting in a certain ambiguity regarding this topic.

This thesis approaches the issue by considering those properties that appear most often in the literature and by further excluding properties that are regarded as “nice-to-have” but which are hard to verify either formally or by experiment. Below, we list the most basic properties on an abstract level and, if appropriate, give details about their peculiarities in the context of SRAM and DRAM PUFs. More precisely, we discuss the aspects of uniqueness, robustness, unclonability and unpredictability.

Uniqueness

In contrast to mathematical functions, the mapping function $\mathbf{PUF}(\cdot)$ is strongly determined by the physical characteristics of the containing object. In particular, the response r depends on the challenge c as well as on the micro- or nanoscale physical structure of the object embedding the PUF. PUF instances that are drawn from the same distribution $\mathcal{P}$ are of the same PUF type. In particular, $\mathcal{P}$ resembles a given manufacturing process. Hence, PUFs that belong to the same underlying distribution $\mathcal{P}$ are the result of the same production technology as well as process and observe similar PUF characteristics. For example, all SRAM PUFs that are extracted from SRAM modules of the same product line are drawn from the same PUF distribution $\mathcal{P}$. By challenging different PUFs that are obtained from the same distribution $\mathcal{P}$ with a fixed challenge c_k, one obtains responses that are highly uncorrelated:

$$\forall \mathbf{PUF}_i, \mathbf{PUF}_j \in \mathcal{P}, \text{ with } i \neq j, \mathbf{PUF}_i(c_k) = r_i, \mathbf{PUF}_j(c_k) = r_j : \mathbf{d}(r_i, r_j) > \epsilon. \tag{3.3}$$

In particular, a PUF is said to be unique, if the distance $\mathbf{d}(r_i, r_j) > \epsilon$ is *high*. Here, $\mathbf{d}(\cdot,\cdot)$ is an abstract distance metric (cf. Section 4.4 for concrete measures).

Robustness

One peculiarity of PUFs lies in the fact that, although a PUF approximates the properties of an injective function, it does not do so perfectly: by challenging a given PUF instance $\mathbf{PUF}_{id}$ repeatedly n times with a fixed challenge c_i, one obtains a set of responses $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ that are similar but not identical, i.e.,

$$\forall r_i, r_j \in \mathcal{R}, \text{ with } i \neq j, \mathbf{PUF}_k(c_i) = r_i, \mathbf{PUF}_k(c_j) = r_j : \mathbf{d}(r_i, r_j) \leq \epsilon. \tag{3.4}$$

Here, ϵ is an upper bound on the dissimilarity of the responses. Usually ϵ is specific to a set of PUF instances that are drawn from the same underlying distribution $\mathcal{P}$. This property of limited robustness emanates from the fact that querying a PUF implies a physical measurement of the underlying object, which usually implicates a certain amount of noise. A PUF is said to be robust if ϵ is *small*.

In case of SRAM-based PUFs, noise phenomena manifest due to those SRAM cells, whose inverters exhibit only a small mismatch in their electrical characteristics. In particular, if the four transistors that form both inverters are highly symmetrical, i.e., the threshold voltages V_{th} of transistors related to inverter A are similar to those related to inverter B (cf. Figure 2.2), the affected cell has no pronounced preference towards one of the two stable states. Hence, the start-up value of the cell "fluctuates" across multiple measurements, which is observed as noise in the PUF measurements. Various techniques to enhance robustness by reducing similarity of threshold voltages of SRAM cells have been proposed, including the enforcement of Negative Bias Temperature Instability (NBTI) [GK14], using Hot Carrier Injection (HCI) to increase the offset of sense amplifiers [BM13] or various approaches that aim at selecting reliable SRAM cells [Eir+12; Xia+14; VDP16] in order to minimize noise. Note however, that invasive techniques like HCI are not applicable to COTS devices due to the high integration of the SRAM module into the MCU die.

In contrast, both DRAM-based PUF constructions, that will be introduced in Section 3.3, leverage the decay process of the charge stored on the DRAM cell's capacitor (cf. Section 2.3.2). Here, the physical phenomena is more complex, as the (stability of the) decay process of a given DRAM cell is not just determined by the analog components of the respective cell, but further by neighboring cells and wires (word- and bit-lines). Hence, noise characteristics of a decay-based DRAM PUF are strongly influenced by the overall physical design of the underlying DRAM module.

Unclonability

PUFs carry the attribute *unclonable* already as part of their name. However, defining "unclonability" is not a straightforward approach. In the literature *physical unclonability* refers to the requirement that it should be hard for anyone, including the manufacturer of the hardware components that embeds respective PUF instances, to produce two identical hardware tokens that implement PUF instances that share an identical function **PUF**(·). In particular,

$$\forall r_i, r_j \in \mathcal{R}, \text{ with } (\mathbf{PUF}_i, \mathbf{PUF}_j) \overset{\$}{\leftarrow} \mathcal{P}, \mathbf{PUF}_i(c_k) = r_i, \mathbf{PUF}_j(c_k) = r_j : Pr(r_i = r_j) \text{ is small.} \tag{3.5}$$

However, in certain application-dependent scenarios, such as authentication this property will not suffice, as some PUF types only possess a small set of challenges $\mathcal{C}$ and responses $\mathcal{R}$ (cf. Section 3.1.2). Using such a PUF as part of a straight forward challenge-response-based authentication protocol, enables a network adversary, who observes the communication between the PUF and a server, to fully model the mapping function **PUF**(·) after he captured all possible challenge-response pairs. Thus, a second variation of this property is denoted as "mathematical unclonability". Here, the requirement lies in the inability of a malicious user, who has unlimited access to a PUF instance, to model the entire PUF functionality, either by exhaustively collecting challenge-response pairs or by successfully simulating the mapping function **PUF**(·) in software. Protection against simulating the mapping function is usually guaranteed, as PUFs must exhibit the additional feature of unpredictability. Instead, protection against observing all possible challenge-response pairs can only be guaranteed by the PUF type itself. Generally, a PUF type that observes a challenge set $\mathcal{C}$ with an exponentially large number of elements, i.e., a strong PUF, qualifies to be mathematical unclonable (cf. Section 3.1.2). However, for a particular class of PUFs that observes an exponentially large set $\mathcal{C}$ and that relies on race conditions of signals traversing circuits (especially Arbiter PUFs) (cf. Section 3.1.2) the property of mathematical unclonability does not seem to hold. Researchers presented modeling attacks based on various machine-learning approaches that exploit the inherent linearity amongst the elements of this PUF type [Rüh+10; HMV12; GTS16], in turn allowing for the construction of a mathematical model of the PUF.

Unpredictability

The unpredictability property refers to a high complexity of the mapping function **PUF**(·). In particular, on the basis of a set of valid challenge-response pairs, it should be hard to infer valid responses from other challenges, without having access to the actual PUF.

Other Properties

Different works of literature propose additional properties of PUFs, some of them regarded as crucial aspects in order to evaluate the PUFs behavior of a given hardware component [RSS09; RBK10; Brz+11; Roe12]. However, this work does not claim to present an exhaustive model for PUFs. Also, some additional properties such as tamper-evidence are not easy to define or to be proven experimentally. Instead, the focus is on establishing a common notation that can later be used for discussing properties of the security protocols and applications, which are constructed on the basis of the PUFs.

3.1.2. PUF Types

Given the diverse PUFs types proposed during the last years, there exist different classification systems that distinguish PUFs regarding various criteria, including the nature of the underlying hardware constructions, the way physical variation is harnessed to obtain a physical measurement or their applicability in security-related protocols. Below, we present the most prevailing approaches to PUF type classification.

Delay-based vs. memory-based PUFs

One way to classify PUFs is to consider the way physical variation of the underlying hardware can be extracted such that analog or digital measurements of this variation can be obtained. In particular for PUF types that are based on silicon chips, i.e., integrated circuits, the two broad categories *delay-based* and *memory-based* PUFs emerged in the literature.

Delay-based PUFs exploit the fact that due to process variation, no two fully identical circuits can be manufactured. Thus, even for a signal traversing two circuits (paths) that share exactly the same layout (number, relative location and type of comprising gates), there will be a small difference in the time it takes both signals to transit the circuits. The resulting time delay of one of both signals is used to extract a PUF response in Arbiter-PUFs [Lee+04; Lin+10; MKP08], Ring-Oscillator (RO) PUFs [Gas+02; SD07].

Memory-based PUFs rely on minuscule differences in physical properties of memory cells. Most of the PUF types that belong to this category exploit metastable states of memory cells that are caused by a mismatch in the physical structure of the otherwise symmetrically laid-out elements (cf. Figure 2.2). These PUF types enforce the metastable state of the memory cells, which eventually settle to one of two possible states that result in an array of logical zeros and ones that can be interpreted as a digital PUF measurement. While most memory-based PUF types rely on variations within memory cells, a few memory-based PUFs exploit different physical aspects, such as data remanence [Hol+12]. In particular, the two novel PUF constructions presented in this thesis [Xio+16; Sch+17a] are based on the decay process of DRAM cells (cf. Section 3.3.1). Prominent examples for memory-based PUFs are SRAM PUFs [Gua+07; H+07], Latch PUFs [SHO07], digital Flip-Flip (DFF) PUFs [MTV08; Van+10], Butterfly PUFs [Kum+08], Flash PUFs [Pra+11; Wan+12], Buskeeper PUFs [SSL12], DRAM PUFs [Teh+15] and Bitline PUFs [HF14].

Extrinsic vs. Intrinsic PUFs

Another approach to classify PUF types focuses on the way the PUF behavior is embedded into a physical object. In the context of PUFs the term *intrinsic* was first coined by Gassend et al. [Gua+07]. It refers to the property that PUF-like behavior is manifested intrinsically during the manufacturing process of a given hardware component. Thus, no modification of the manufacturing process and no post-modification of the assembled hardware components is required to establish a PUF. Later, Maes [Roe12] extended this definition to also require that the evaluation of the PUF is conducted internally, i.e. the PUF response is produced inside the containing hardware. The advantages of intrinsic PUFs are manifold. As intrinsic PUFs are created using standard manufacturing processes, they introduce no overhead during production. Since they can be provided “at zero-cost”, i.e., they are an artifact of the production process, they exhibit high dissemination. Combining both advantages, intrinsic PUFs are particular interesting for low-cost devices, which are produced under the premise of high profit.

Accordingly, extrinsic PUFs are constructed in a dedicated process as the physical mechanisms that exhibit PUF behavior require customized hardware layouts that, by definition, are not part of standard hardware components. Hence, they are either synthesized as Application-Specific Integrated Circuit (ASIC) or they are implemented on Field-Programmable Gate Arrays (FPGAs). Most delay-based PUFs are extrinsic PUFs, including Arbiter-PUFs and RO-PUFs. In accordance with the definition of Maes [Roe12], extrinsic PUFs further require external equipment (i.e., which is not part of the object embedding the PUF) for measuring PUFs responses.

Weak vs. Strong PUFs

From a security point of view, an important discrimination factor for PUFs is the size of their space of challenges $\mathcal{C}$ and responses $\mathcal{R}$. The notion of weak and strong PUFs was first introduced by Guajardo et al. [Gua+07] and was later improved by Rührmair et al. [RBK10] A strong PUF exhibits a set of challenges (and responses) with a number of elements that is exponentially large in the number of hardware components, constituting the PUF. Given this property, an attacker with access to the PUF for an unlimited time, will not be able to exhaustively query the PUF in order to create a complete model of the PUF behavior (cf. Section 3.1.1). In turn, weak PUFs are prone to mathematical clonability of their challenge-response space. In fact, at the time of writing this thesis, all intrinsic PUFs, including the PUF types considered in this work, are considered as weak PUFs and hence do not exhibit mathematical unclonability. However, a common approach to compensate this constraint is to introduce a controlled interface for querying the PUF at system level and hence to create a so- called controlled PUF [Gas+07]. Strong PUFs are generally qualified to be used for authentication purposes, whereas due to the small number of challenges, weak PUFs are usually considered for secure key storage (cf. Section 3.1.3). Nevertheless, in Chapter 7, we will show how a lightweight secure authentication protocol can be constructed on the basis of the decay-based DRAM PUF presented in Chapter 3.3.1, which provides a number of challenge-response-pairs, which is nearly linear in the amount of DRAM cells.

3.1.3. Applications of PUFs

Due to their favorable properties discussed previously, PUFs have been proposed as a hardware-based security primitive for various security protocols and applications. Below, we will give an overview of the numerous usage scenarios of PUFs and reference their related work in the literature.

Secure Key Storage

The interconnection of embedded devices requires the secure identification of the involved platforms, used to exchange potentially sensitive data. Using secure device identification, impersonation attacks during the communication of the involved devices or hardware counterfeiting can be thwarted. Commonly, a unique identifier is stored on the respective embedded platform for the purpose of secure identification. A basic approach is to include a unique serial number or a shared secret key to the hardware component at manufacturing time [Smi10]. More advanced devices that provide dedicated secure hardware usually rely on public-key cryptography and signatures to establish authenticity and hence to identify a device (cf. Chapter 7). On embedded devices without such secure facilities, a common approach to device identification is to embed cryptographic keys in each device by burning them in at manufacturing time, i.e., using masked Read-Only Memorys (ROMs). However, this solution comes with potential pitfalls, such as increased production complexity as well as rather limited protection against key extraction attempts [Arm+10]. In particular, long-term secrets are vulnerable to key extraction, given that they permanently exist in digital form and since most NVMs can be attacked by means of (semi-) invasive attacks [TW11]. From a security perspective, the approach of using traditional NVM-based key storage becomes even more unattractive, due their high ubiquity and hence the increased physical attack surface of such embedded platforms.

As an alternative, researchers have proposed PUFs for secret key storage [TŠ06; Tuy+07; SD07], as they are a lightweight means to provide a device-unique fingerprint. Moreover, by using PUFs cryptographic keys are of ephemeral nature; they only exist for a limited and usually very short time. In a PUF-based key storage, a key is created "on-the-fly", i.e., after the PUF was challenged for this key. While the PUF-enabled device is turned off, the key is not existent in any form. This volatile nature of a long-term secret heavily decreases the attack surface and complicates key extraction attempts. Especially for intrinsic PUF types secure key storage is a preferred use case, as they provide inherent hardware identities without additional costs and hence avoid the addition of secure non-volatile key compartments. Furthermore, the process of explicitly burning keys into NVM is eliminated. Thus, SRAM-based PUFs, belonging to the class of intrinsic PUFs, are mainly used for secure key storage [Gua+07; Sel+11].

However, PUF measurements always contain a certain amount of noise. In order to extract a stable output from a noisy PUF measurement X that can be used to derive a cryptographic key K, a Fuzzy Extractor (FE) construction [DRS04] can be applied. The FE maps the noisy PUF measurement X to a stable message by compensating the noise, usually by means of Error-Correcting Codes (ECCs). Commonly, FEs work in two phases, a generation phase `FE.Gen()` performed upon enrollment at a trusted party (i.e., the manufacturer or the system integrator) and a reconstruction phase, conducted by the PUF user, during which a process `FE.Rec()` is performed after each PUF measurement. During `FE.Gen()` a secret key K and a public Helper Data W are derived from a noisy PUF reference (enrollment) measurement X. The algorithm F*E.Rec*() decodes a noisy PUF measurement X' back into the key K, thereby using Helper Data W. This works as long as X and X' are close enough (e.g., both are two PUF responses to the same challenge obtained from the same PUF instance). The enrollment and reconstruction processes are depicted in Figure 3.1.

Hardware-Software Binding

One of the most tempting scenarios is the illegitimate reproduction of embedded systems, where an adversary reproduces existing devices by copying their firmware to counterfeit, cheaper hardware. By selling those cloned devices, adversaries cause financial loss for the manufacturer of the original system. As embedded devices usually lack the implementation of trusted hardware such as TPM, Intel SGX or ARM TrustZone (cf. Chapter 2.2), intrinsic PUFs depict an efficient and

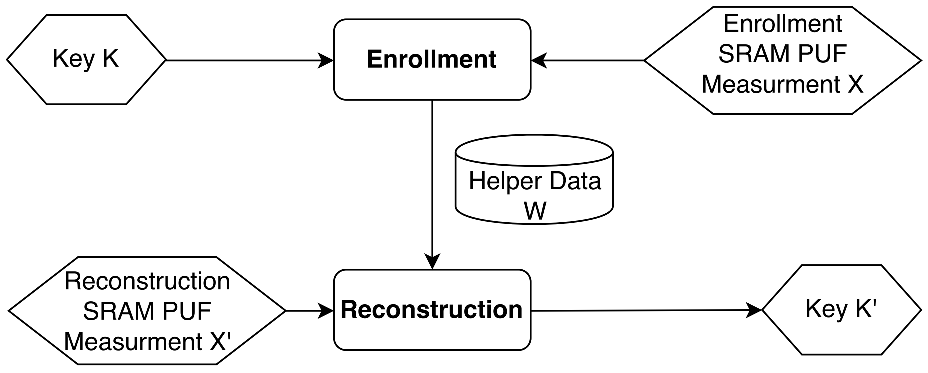

Key K
Enrollment
Enrollment SRAM PUF Measurment X
Helper Data W
Reconstruction SRAM PUF Measurment X'
Reconstruction
Key K'

Figure 3.1.: Illustration of the enrollment and reconstruction phases of a Fuzzy Extractor.

widely-available solution to this problem. Especially SRAM-based PUFs have been considered as a primitive to bind firmware to the underlying hardware [GMS09; Gua+07; LMA16; LT13].

In this thesis, we present two solutions that address the problem of protecting firmware on embedded devices against firmware extraction attempts. A first solution presented in Chapter 5 proposes a secure boot scheme that protects the firmware at boot time by leveraging the PUF characteristics of on-chip SRAM on MCUs [Sch+14] in order to decrypt the firmware prior to execution. A second solution presented in Chapter 6 protects the subsequent firmware execution stage by intertwining the execution of a firmware with the underlying hardware [KSK15]. Again, we rely on an SRAM-based PUF extracted from commercial embedded devices and combine it with the concept of self-checksumming hashes to provide a hardware-based security anchor.

Secure Authentication

Based on the idea of deriving device-unique identifiers from PUF-enabled hardware components, PUFs have been further proposed to be used as building blocks of authentication protocols [**tuyls2006rfid**; ÖHS08; SVW10; SD07; Van+12]. Most solutions consider a low-cost yet PUF-enabled client that is authenticated against a powerful server. A classic approach towards authentication foresees an enrollment procedure that results in a challenge-response-pair database stored at the server. During the actual authentication process the server challenges the client to provide a valid response. It is noteworthy that in traditional authentication protocols only strong PUFs (cf. Section 3.1.2) can be used. This is due to the fact that the numerous possible challenges mitigate replay attacks (challenge-response pairs cannot be reused).

However, in Chapter 7 we present a novel authentication protocol that also enables authentication in the presence of a PUF with only a small number of challenge-response pairs. We do this by exploiting the special decay characteristics of DRAM cells as part of a novel PUF construction called the decay-based DRAM, which is introduced in this thesis in Chapter 3.3.1.

Device Attestation

A major challenge in computer security is to establish the trustworthiness of remote platforms. Remote attestation is the most common approach to this challenge. It allows a remote platform to assess and report its system state in a secure way to a third party. Recently, PUFs have been employed as a building block in attestation schemes focusing on low-cost and resource-constrained devices [Koç+11; Kon+14; SSW11].

Unfortunately, existing attestation solutions either provide low security, as they rely on unrealistic assumptions, or are not applicable to commodity low-cost and resource-constrained devices, as they require custom secure hardware extensions that are difficult to adopt across IoT vendors. In Chapter 8, we propose a novel remote attestation scheme, named Boot Attestation, that is particularly optimized for low-cost and resource-constrained embedded devices. We achieve a high coverage of existing commercial devices by relying on standard hardware components, such as SRAM-based PUFs, in order to support implementation of the Boot Attestation protocol even on the most low-end IoT platforms available today.

Random Number Generation

Many cryptographic primitives rely on random data to ensure security. The generation of keys, salts or nonces requires random data to be unpredictable to attackers. Therefore, Random Number Generators (RNGs) need to be fed with high entropy seeds. In order to generate random numbers for cryptographic applications, ideally a physical source should be harnessed which provides true randomness. Such non-deterministic sources derive their randomness from underlying physical properties that exhibit unpredictable behavior. However, there are two important drawbacks to most of these physical RNG constructions. First, they require specific hardware to extract randomness from the physical entities on the device. Second, the throughput of such RNGs is often too low for cryptographic applications, where large streams of random bits are required. Thus, PUF-based RNGs have been proposed, that exploit the noise inherent in a PUF measurement in order to provide truly random data [HBF09; Lee+12; Van+13].

3.2. SRAM-based PUFs

It has been shown that selected SRAM modules show PUF-like behavior [H+07]. Further research in this area supports the applicability of SRAM as a Physically Unclonable Function [MTV09; SL12]. Using SRAM as PUFs exploits manufacturing variations, which manifest themselves in a bias of SRAM memory cells. During the power-up phase SRAM cells are drawn from a metastable state to one of their bistable settings, representing either the logical value of zero or one (cf. Section 2.3.1). The tendency to settle towards a stable state is due to minuscule deviations from the otherwise symmetrical layout of the transistors that form two cross-coupled inverters. On a logical level, the inverters establish a positive feedback loop that enforces exiguous differences in the transistors so that the SRAM cell leaves the metastable state, eventually resulting in a so-called start-up value of one of the logical values. In doing so, most cells show a stable start-up behavior, i.e., a strong tendency to initialize to a fixed start-up value. An array of multiple start-up values of SRAM cells creates a start-up pattern, which is evaluated as a PUF measurement X, which in turn serves as the fingerprint for the SRAM module.

In order to retrieve a measurement from an SRAM-based PUF, it is required to access the unmodified start-up pattern of the SRAM cells. Here, the attribute *unmodified* is particularly stressed. It refers to the state of the SRAM array shortly after power-up and before any process performed a write-access to the memory. Any process of writing to the SRAM would most likely alter the start-up pattern in such a way that the resulting values are an artifact of the corresponding software process, instead of the underlying physical characteristics of the SRAM module. The logic to query an intrinsic SRAM-based PUF consists of the following steps:

a) Setting up a communication interface to transmit the start-up values to a host machine, usually leveraging Univeral Asynchronous Receiver Transmitter (UART).

b) Determining start and size of the memory range to be queried as a PUF. During initial evaluation this region is to be maximized to get an exhaustive characterization of the SRAM module. After evaluation of the PUF instance however, only a minimal SRAM region will be used, i.e., to reconstruct an 128 bit symmetric key, to consume as little SRAM as possible.

c) After determining the PUF size, reading the individual bytes of the corresponding memory region and sending them over to the host machine for evaluation or as input to a Fuzzy Extractor construction for key derivation.

d) For security reasons, the start-up pattern of the respective PUF is overwritten after being read, in order to be inaccessible to other processes.

In case a dedicated SRAM chip can be leveraged for PUF usage [Int17], the entire SRAM is available for extracting a fingerprint. In this case and depending on the design of the power supply of the SRAM module, the SRAM can be powered and hence the PUF can be queried at any given time. However, leveraging intrinsic on-chip SRAM of commercial devices as done in this thesis imposes several critical limitations that influence not just the means of interfacing the PUF but also the time at which the PUF can be accessed. In this case of leveraging an SRAM module that is not exclusively dedicated to PUF usage but primarily serves as the main memory of the embedding MCU, special care must be taken to make sure that unmodified SRAM start-up patterns are extracted. In particular, the PUF measurement must be conducted right after powering the SRAM module, which usually corresponds to an early stage during the boot phase of the entire device. Furthermore, only a fraction of the SRAM can be used, as the rest must be kept available for firmware execution, i.e. stack, heap, etc. and hence may have been modified before PUF measurement.

In order to maximize the fraction of unmodified SRAM start-up values, on some platforms the interface to the PUF can be implemented leveraging working and configuration registers as storage for variables and pointers, by means of Assembly. In this way, no or little SRAM is consumed by the PUF interface itself. However, this approach is usually only possible on very low-end platforms, such as 8 bit processors, that exhibit a less complex structure and hence require less effort to initialize the device. Furthermore, more complex SoC-based platforms implement NAND flash modules, which in contrast to NOR flash (generally part of low-end MCUs) are not capable of Execute-In-Place (XIP), i.e., executing code. Hence, on NAND-based platforms the firmware is usually first copied to SRAM, in turn making large portions of the memory unavailable for PUF-usage. In order to visualize this issue, Figure 3.2 shows unmodified SRAM on a TI Stellaris MCU implementing NOR flash. In contrast, Figure 3.3 shows an enrollment of the SRAM values on a PandaBoard, implementing NAND flash. Here, large portions are unavailable for PUF usage, as the firmware is copied to SRAM upon boot. In both figures each black pixel represents a zero bit, whereas white pixels represent bits of value one.

In general, SRAM PUFs resemble a weak, intrinsic memory-based PUF type. Subsequently, the property of mathematical unclonability does not hold in the presence of a well-equipped invasive attacker [Hel+13]. Such an attacker is able to retrieve the start-up values of an SRAM chip in order to create a PUF model and subsequently to clone the PUF. In one attack type, the attacker does this by first leveraging the side-channel which emerges due to photonic emission, in order to derive the characteristics of the start-up values of a given SRAM module. In a next step, a second SRAM module is modified using a focused ion beam laser, in such a way that

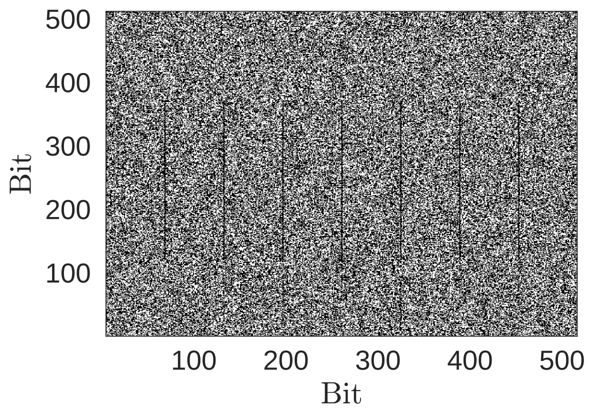


Figure 3.2.: Bitmap of 32 kB SRAM start-up values extracted from a Texas Instruments LM4F120H5QR development platform.

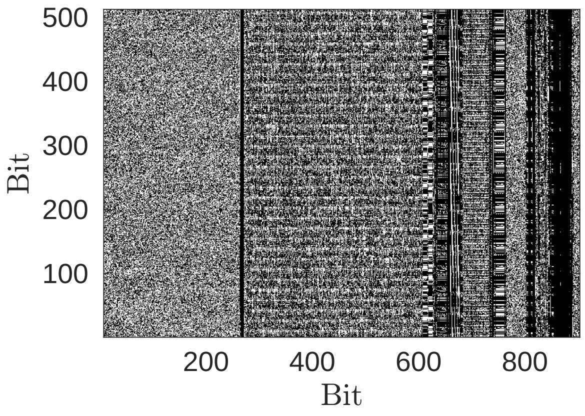


Figure 3.3.: Bitmap of 56 kB SRAM start-up values extracted from a PandaBoard ES SoC. Only the smaller left portion exhibits unmodified start-up values, whereas the major part of the bitmap has been pre-initialized by ROM code.

corresponding transistors are disabled, to mimic the start-up behavior of the SRAM module to be cloned. However, it is unclear whether such invasive attacks are successful against modern SoC platforms that are manufactured using a PoP packaging technology, where functional blocks of the SoC, including the SRAM module and the CPU are stacked above each other. Due to the stacked layout, an ion beam laser that is applied from the bottom of the package would first disable the CPU, in turn destroying the entire SoC and the PUF instance.

3.3. DRAM-based PUFs

The earliest approach to exploit manufacturing variations of DRAM cells for identification and random number generation was reported in [Ros+13a; Ros+13b], where an embedded DRAM chip was designed to generate fingerprints to mitigate hardware counterfeiting. In subsequent work, through a memory controller in FPGA, Keller et al. [Kel+14] proposed to use the decay of external DDR3 modules for extracting random bits as well as unique identifiers. Lui et al. [Liu+14] evaluated the uniqueness, robustness and min-entropy of external DRAM modules using a FPGA setup, and proposed a secure key storage scheme based on DRAM modules. Hashemian et al. [Has+15] designed a circuit exploiting the variation of the write reliability of DRAM cells, and presented an authentication scheme based on signatures generated. Rehmati et al. [Rah+15] made use of the error pattern in approximate DRAM as a system fingerprint. Tehranipoor et. al [Teh+15] used startup values of DRAM cells to create a device signature. In the previous works, either dedicated circuits were designed or FPGAs were used. In contrast, the DRAM PUF proposed in the following section is the first intrinsic DRAM PUF that was extracted from Commercial Off-The-Shelf devices. Furthermore, the decay-based DRAM PUF and the Rowhammer PUF are the first DRAM-based PUFs that can be challenged during runtime in commodity hardware.

3.3.1. The Decay-Based DRAM PUF

As mentioned in Section 2.3.2, DRAM cells require a periodic refresh of the stored charges, as otherwise the capacitors lose their charge over time which is referred to as DRAM cell *decay* or *leakage*. As depicted in Figure 3.4, charge may leak via neighboring analog components, such as wires, capacitors, etc. The hardware memory controller takes care of a periodic refresh, whose interval is defined by the vendor, and is usually between 32 ms and 64 ms. Without this periodic refresh, the logical values stored in the cells may flip to their opposite values.

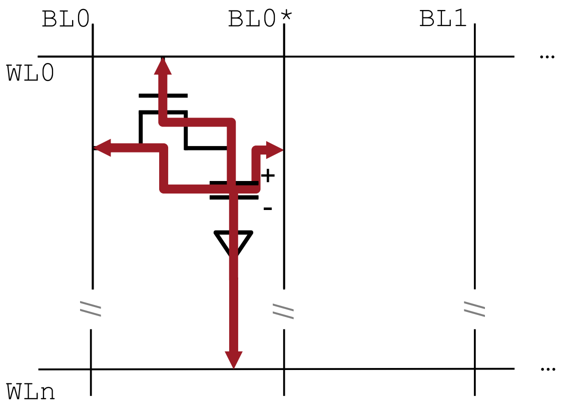


Figure 3.4.: A single DRAM cell consists of a capacitor and a transistor, connected to a word-line (WL) and a bit-line (BL or BL*). Red arrows indicate potential leakage paths for dissipation of charges that lead to PUF behavior.

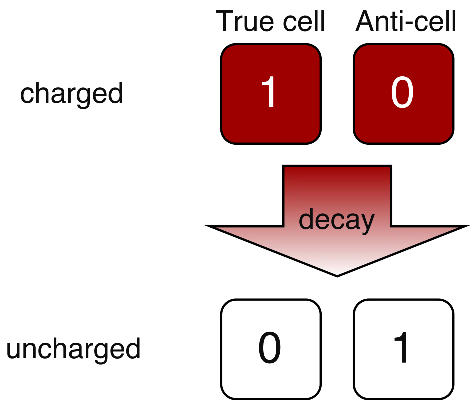


Figure 3.5.: Decay process of true and anti-cells. Red cells depict charged capacitors whereas white cells are uncharged.

To be precise, if all cells have been initialized to their charged state, logical values of true cells decay to ‘0’, while anti-cells decay to ‘1’, as shown in Figure 3.5 (cf. Section 2.3.2). Because of the manufacturing variations among DRAM cells, some cells decay faster than others. The unique decay characteristics of individual DRAM cells can be exploited for a decay-based DRAM PUF, as it will be shown in this thesis.

The process of exploiting the unique decay behavior of DRAM cells in order to extract a PUF measurement X is summarized in Figure 3.6. The starting point (a) comprises the DRAM module being configured for ordinary use, where the memory controller periodically refreshes the entire cells’ content. In a first step (b), the PUF memory region, defined by starting address (`addr`) and size (`size`), is reserved so that it does not contain any user-space or Operating System (OS) programs. This region is depicted as a shaded gray rectangle in the figures. Furthermore, the refresh for the PUF region is disabled and the initialization value (`iv`) is written to the region. Next, (c) for a given decay time (`t`), the memory region containing the PUF is not accessed to let the cells decay. After the decay time has expired, (d) the memory content is read in order to extract the PUF measurement which manifest itself in the unique location of the flipped bits. At the end, (e) the normal operating condition of the memory is restored and the memory region is made available to the OS again. Memory regions within a DRAM module that are used for obtaining PUF measurements are called *logical DRAM PUFs*. For a particular DRAM, each logical PUF is determined by the following parameters: (i) `addr`, the starting address of the logical PUF,

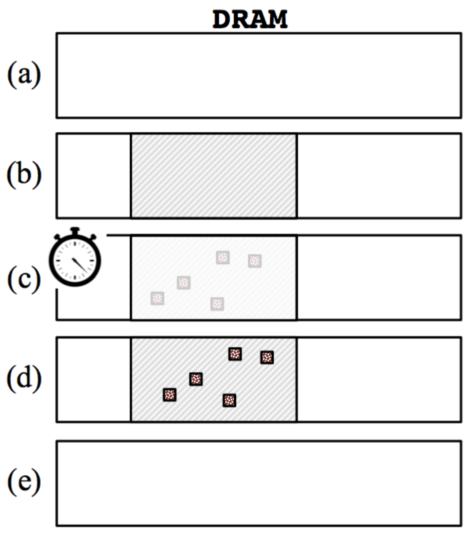


Figure 3.6.: Process of querying a decay-based DRAM PUF instance.

and (ii) `size`, its size, as discussed above. A typical DRAM memory module can then be divided into thousands or more logical PUFs.

Two additional parameters are needed to define a DRAM PUF challenge. First, an initialization value (`iv`) needs to be chosen, written to the DRAM cells before any decay process starts. Second, the desired decay time (`t`) has to be specified. After the decay time has expired, enough charge has leaked from some cells so that their stored logical bits has flipped. As the decay time and the positions of the flipped bits are unique for individual DRAM regions, the "pattern" of decayed (flipped) bits, for a given decay time `t` serves as the PUF response.

In order to derive a cryptographic key from the PUF response using a minimum number of DRAM PUF cells, the entropy within a logical DRAM PUF response needs to be maximized. The value stored in a DRAM cell before it decays, `iv`, plays an important role, as true cells decay to '0' and anti-cells decay to '1'. Thus, for example, if a true cell is initialized to '0', the decay effect cannot be observed. If the physical layout of the DRAM module is known (i.e., the distribution of true cells and anti-cells, and hence the individual decay directions), it is possible to construct an initialization value that maximizes the number of observable bit flips in the PUF response.

However, the physical layout is rarely known. Furthermore, the optimal initialization value would need to be part of the challenge, or it would have to be stored on the device. In the evaluation presented in Chapter 4, we use a fixed initialization value `iv` of '0' for all cells within the memory being used as PUFs. Thus, the entropy of our measurements can further be improved if the initialization value is varied so that each cell is initialized with a logical value that corresponds to a state, where charge is stored on the cells' capacitor (i.e., '1' for true cells, and '0' for anti-cells).

Overall, the challenge of a DRAM PUF consists of the tuple ($\mathbf{PUF}_{id}$,`t`), where $\mathbf{PUF}_{id}$ denotes the logical PUF instance (`addr` and `size`) and `t` denotes the decay time after which the memory content is read. In our experiments we fixed the value of `iv`, hence we do not specify the parameter explicitly.

3.3.2. The Rowhammer PUF

The Rowhammer PUF is an intrinsic memory-based PUF that relies on DRAM decay characteristics and leverages the Rowhammer (RH) effect in DRAM modules [Sch+17a]. More precisely, the Rowhammer PUF induces bit flips in DRAM due to rapid and repeated access of DRAM rows, whose location creates a device-unique fingerprint.

The Rowhammer effect was first exposed by Kim et al. [Kim+14] by repeatedly and rapidly accessing DRAM wordlines in order to induce bit flips in adjacent DRAM locations. Using an FPGA testbed the authors conducted extensive experiments and showed the prevalent existence of disturbance error in various commodity DRAM chips. A recent report demonstrated the existence of DRAM disturbance errors in recent DDR4 [Lan16] modules as well.

Prior research has mainly focused on Rowhammer-based attacks, to implement various practical security exploits. In [SD15] and [Vee+16], the authors rely on the Rowhammer effect to gain root privileges by flipping bits in Page Table Etries (PTE). Xiao et al. [Xia+16] use Rowhammer in cross-virtual machine settings in order to attack Xen's paravirtualized memory isolation by employing Rowhammering from within a malicious virtual machine. In [Raz+16] and [BM16], authors successfully attack RSA by creating bit flips in keys stored in DRAM. The authors of [Aic15] and [Awe+16] showed that ECC and doubled DRAM refresh rates are not enough to mitigate the Rowhammer attack.

In contrast, in this thesis the Rowhammer effect is used for the first time in a *positive context*, i.e., to design a novel PUF. Leveraging the Rowhammer effect as a promising candidate for a PUF is motivated by previous work that has shown that the locations of disturbance errors in DRAM cells are repeatable [Kim+14; Vee+16].

However, the number of bit flips introduced by the Rowhammer effect can be relatively small, and thus may only provide a limited amount of entropy. We thus introduce three techniques to help increase entropy without changing the DRAM physical properties.

- First, we disable DRAM refresh for those memory locations where the PUF is located. This prevents the PUF cells from being recharged, as would happen if normal refresh was on, and increases the number of bit flips.
- Second, multiple DRAM rows are used together to create an instance of the Rowhammer PUF that encompasses larger amount of cells.
- Third, hammering time and initial values of the DRAM cells are controlled to induce a maximum number of bit flips.

There are many parameters that can influence the Rowhammer PUF. In the following section we describe such parameters that influence the number and location of bit flips, i.e. the general PUF-behavior, the most. In Section 4, we present an evaluation upon which the most suitable values among these parameters are selected.

`RH type`: As presented in the literature [Awe+16; SD15], there are two approaches in order to induce the Rowhammer effect. If for one row, which will be part of the PUF measurement (the *PUF row*) there is only one adjacent row which is repeatedly read to induce bit flips (the *hammer row*) we speak of single-sided Rowhammer (`SSRH`). In contrast, double-sided Rowhammer (`DSRH`) involves exposing both neighbors of a particular PUF row as hammer rows. The patterns of hammer and PUF rows, used to conduct `SSRH` and `DSRH`, are shown in Figure 3.7.

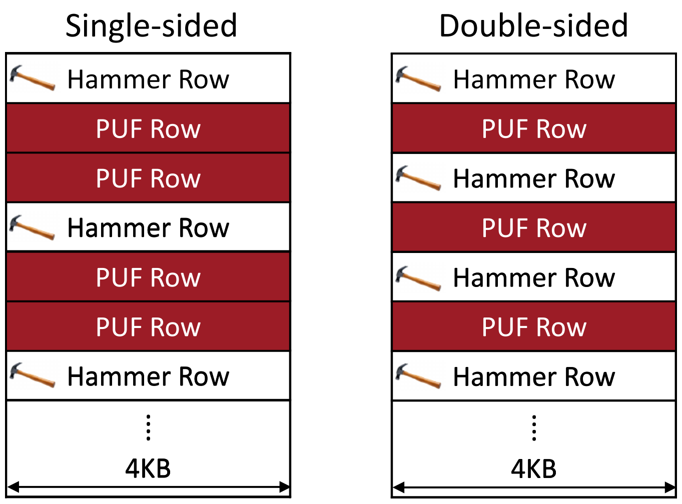


Figure 3.7.: Layout of hammer rows and PUF rows due to the `RH type` parameter. Left: single-sided (`SSRH`). Right: double-sided (`DSRH`). A row size of 4 kB is assumed.

`Hammer row iv`: For the memory range that corresponds to `addr` and `RH size`, each included hammer row will be pre-initialized by writing the initial value `Hammer row iv` to it, before conducting the Rowhammer process.

`PUF row iv`: Similarly, all the PUF rows that are included in this memory range are initialized with the initial value `PUF row iv` prior to the Rowhammer process. Both, `Hammer row iv` and `PUF row iv` are important parameters as disturbance errors are caused by interaction of DRAM cell charges. Moreover, as discussed in Chapter 2.3.2, DRAM cells represent a particular logic value using different charge states, resulting in true cells and anti-cells. Consequently, initializing a true cell with '0' would not allow to observe a bit flip after its charge has leaked. Thus, it is important to evaluate the effect of different values of `PUF row iv` and `Hammer row iv`.

`RH time`: The Rowhammer time `RH time` defines the total duration of the PUF measurement, including disabling the refresh rate and conducting the hammering process. `RH time`, just as `RH size` and `RH type`, affects how many times each hammer row will be accessed in total.

Given the above parameters, the process of accessing a Rowhammer PUF is depicted in Algorithm 1. Based on `PUF address`, `PUFsize`, and `RH type`, the DRAM region for the Rowhammer PUF is defined. First, this DRAM region is reserved, so that no other program accesses the same region. Next, the PUF rows and hammer rows are initialized by `Hammer row IV` and `PUF row IV`, respectively. The PUF query is started by disabling the DRAM auto-refresh in the next step. This is done by using the same technique as employed in [Xio+16]. Subsequently, the process of repeated hammering of rows is started. For this purpose, the hammer rows need to be accessed repeatedly for a certain time. This is achieved by a read operation to the first word of each hammer row. Hence, bits in the PUF rows will start to leak charge and will eventually flip. After `RH time`, the process ends and the DRAM refresh is enabled again. Finally, the PUF measurement can be read from the PUF rows.

Because the Rowhammer PUF is inherently tied to the underlying DRAM, there are three factors that can influence the operation of the PUF:

Temperature: Prior work has shown that victim cells are not strongly affected by temperature [Kim+14]. However, the Rowhammer PUF is a joint effect of the Rowhammer effect and DRAM decay. Thus, we evaluate the temperature effect in Section 4.5.3, which confirms that

Algorithm 1: Process of the Rowhammer PUF-query.

```
input : RH type, addr, RH size, Hammer row iv, PUF row iv, RH time
output: PUF measurement X
1 reserveMemory(addr,RH size);
2 initializePUFRows(PUF row iv);
3 initializeHammerRows(Hammer row iv);
4 disableAutorefresh();
5 while $t <$ RH time do
6     for $r_i \in$ hammer R do
7         read($r_i$);
      end
  end
8 enableAutorefresh();
9 X =readPUFRows();
```

the Rowhammer PUF exhibits increased bit flips and stable noise values at higher operating temperatures.

Voltage: Prior work has also shown that voltage affects the leakage in DRAM cells [HSS98]. In commodity, off-the-shelf devices there is currently no interface to control the voltage of DRAM cells. We assume that for the Rowhammer PUF, the DRAM operates at the factory specified voltage parameters. Voltage factors will be investigated in future work.

Error-Correcting Code (ECC): ECC can be used in DRAM to protect from rare bit flips. Many computing platforms, such as the PandaBoard used in this work, do not have ECC implemented. Even if ECC is present, the authors of [Aic15] showed that ECC is not enough to mitigate the Rowhammer effect. In order to use the Rowhammer PUF when ECC is used, the PUF size would have to be increased. Further, ECC registers that indicate rows, which observed bit flips, could potentially be exploited for PUF measurements. We will explore this in future work. In this thesis we assume that no ECC is used.

3.4. Chapter Summary

As shown in this chapter, the favorable properties of PUFs are used in manifold security-related applications. Whilst existing works in the literature mainly dealt with extrinsic PUFs that are either synthesized in FPGA or ASIC, this work focuses on *intrinsic* memory based PUFs. Furthermore, three different intrinsic PUF types were introduced, which will be used in the remainder of this work. In particular, we discussed the functionalities of the SRAM-based PUF and introduced two novel PUF types based on DRAM, the decay-based DRAM PUF and the Rowhammer PUF. In the next chapter, intrinsic PUF instances that are manifested on low-cost COTS MCUs are identified, extracted and subsequently evaluated towards their PUF characteristics.

Chapter 4

Empirical Analysis of PUF Instances

Published Content

- Wenjie Xiong, André Schaller, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Run-time Accessible DRAM PUFs in Commodity Devices.* In: *International Conference on Cryptographic Hardware and Embedded Systems.* 2016, pp. 432–453.
- André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Intrinsic Rowhammer PUFs: Leveraging the Rowhammer Effect for Improved Security.* In: *IEEE International Symposium on Hardware Oriented Security and Trust.* 2017, pp. 1–7.

As shown in the previous section, Physically Unclonable Functions (PUFs) have favorable properties that make them an interesting building block for a number of lightweight security applications. However, in order to make use of them on Commercial Off-The-Shelf (COTS) Microcontroller Units (MCUs), PUF behavior must be observable on such devices in the first place. The mere presence of a hardware component that is known to observe PUF-like behavior is not a strong guarantee that this component indeed yields a PUF instance[1]. Hence, a potential hardware component must be tested towards its property to observe basic PUF-like behavior at all. Moreover, potential PUF instances must be accessible so that no hardware modifications are required. Furthermore, any code that implements an interface to an intrinsic PUF must be compatible with the existing software stack and must meet the rather strict underlying requirements of the platform. In particular, most low-cost device types limit the size of the boot loader to be in the range of a few kilobytes. Especially in the case of SRAM-based PUFs, which rely on the “freshness” of the SRAM start-up values, the PUF interface must hence be part of the boot loader, which can display an engineering challenge due to strict memory constraints. Furthermore, the code that interfaces the PUF can render parts of the start-up values useless, as it is executed from the same SRAM module, which is also leveraged for PUF usage, as shown in Chapter 3.2.[2] Moreover, some device types execute vendor-specific routines as the very first code after start-up, which pre-initializes, i.e., overwrites the SRAM and hence destroys any entropy, which otherwise could be used to extract a device unique key [VBN15]. Lastly, while the mere existence of PUF-like behavior is necessary in order to leverage a PUF in security-critical

[1] During our evaluation we came across a few embedded systems, whose Static Random-Access Memory (SRAM) start-up values did not contain any robust SRAM cells that could be used for identification.

[2] This situation is the standard setup for the targeted device class that features on-chip memory modules, i.e., MCU-based devices, as SRAM modules are not exclusively dedicated to PUF usage.

scenarios, it must further be guaranteed that it exhibits reliable characteristics, guaranteeing the security of the overlying protocol, even under changing ambient conditions or long time usage. In order to obtain statements about the properties of a given PUF instance, exhaustive evaluation needs to be conducted, prior to its actual deployment.

As we will show in this chapter, various intrinsic memory-based PUF types can indeed be found in standard memory components of various embedded COTS platforms. After introducing software-only means to query these PUF instances, this chapter presents empirical evaluation results of the three PUF instances introduced in Chapter 3, namely the SRAM- and Dynamic Random-Access Memory (DRAM)-based PUFs, as well as the Rowhammer PUF. For this purpose, we assessed a wide range of different embedded device classes, ranging from low-end 8 bit MCU-based devices, such as the PIC16F1825, to more sophisticated System on Chip (SoC) platforms, like the PandaBoard. We strove for a very diverse set of device types, all of which do not implement dedicated secure hardware, including platforms that implement ARM, PIC, AVR and x86 cores. In this way, we are able to validate that a high number of embedded COTS devices feature on-chip memory modules that can be used to extract instances of respective PUF types. For this purpose, we evaluated such commodity platforms that exhibit high dissemination in real-life operational areas and hence are representative, wide-spread device types.

Note, that parts of the evaluation section that discusses results of SRAM-based PUFs (cf. Section 4.5.1) consist of results that were obtained during a joint project. Those results have been published previously as part of the doctoral thesis of Anthony van Herrewege [Van15]. Hence, the SRAM evaluation results presented in this thesis are an extension of the former publication and are enhanced by the following content:

- Evaluation of additional device types, namely the STM32F100RB, LM4F120H4QR and the PandaBoard with respect to SRAM-based PUF instances and
- inter-device min-entropy results and
- Context Tree Weighting (CTW) evaluation results.

Moreover, this chapter also presents evaluation results of the DRAM-based PUF types, which were introduced in this thesis, i.e., the decay-based DRAM PUF and the Rowhammer PUF.

4.1. Evaluation Goal and Approach

The goal of the evaluation is heavily driven be the applications that build on the PUF instances. At its core, each PUF-enabled application, which will be presented in the following chapters, relies on a secret key, derived from an intrinsic memory-based PUF. Therefore, we analyze the PUFs for their *robustness*, *uniqueness* and *entropy* characteristics (cf. Chapter 3.1). Additional tests under exceptional ambient conditions, i.e., temperature and time stability, have been conducted to prove the robustness of the PUF instances in varying real-life environments. During evaluation we excluded tests that vary voltage or voltage-related aspects, i.e., variation of supply- or threshold-voltages or ramp-up times). This is motivated by the fact that COTS devices usually do not provide interfaces to alter voltage parameters.

The evaluation results are necessary for enabling PUF-based security protocols and applications on COTS MCUs under test. We follow an empirical evaluation approach that relies on testing

Device Type	Processor	Manufacturer	Device Class	Memory Size	Devices	Measurements
			SRAM-based PUF			
PIC16F1825	PIC	Microchip	Class-0	1 kB	16	3700
ATMega328p	AVR	Atmel	Class-0	2 kB	16	9695
MSP430F5308	MSP430	Texas Instruments	Class-0	6 kB	15	3174
STM32F100R8	ARM Cortex-M3	STMicroelectronics	Class-1	8 kB	11	3419
STM32F100RB	ARM Cortex-M3	STMicroelectronics	Class-1	8 kB	14	1069
LM4F120H5QR	ARM Cortex-M4F	Texas Instruments	Class-1	32 kB	15	1000
PandaBoard (ES)	OMAP4460	Texas Instruments	Class-2	56 kB	16	1000
			DRAM: decay-based PUF			
Intel Galileo	Intel Quark X1000	Intel	Class-2	256 MB	5	50
PandaBoard (ES)	OMAP4460	Texas Instruments	Class-2	1 GB	4	50
			DRAM: Rowhammer PUF			
PandaBoard (ES)	OMAP4460	Texas Instruments	Class-2	1 GB	4	20

Table 4.1.: Device types used for evaluating characteristics of different PUF types.

a finite number of devices regarding their PUF characteristics to derive statements about PUF behavior, in contrast to a modeling-based approach.

4.2. Evaluated Device Types

In order to show the ubiquitous existence of memory-based intrinsic PUF instances, especially in on-chip components of low-cost COTS MCUs, we evaluated a broad range of diverse platforms, including low-cost 8 bit MCUs up to more complex SoCs. Furthermore, while the majority of tested platforms implement an ARM processor, we also included an x86-based platform as part of the analysis of the decay-based DRAM PUF. Table 4.1 gives an overview of the device types under test and their hardware properties. Note that the number of measurements taken for evaluation of the DRAM-based PUF types is significantly smaller. The is due to the nature of the process of querying these PUF instances: As shown in Section 3.3, both DRAM-based PUF types leverage a decay-process over time, in order to produce a PUF measurement. Hence, generating a comprehensive dataset that considers various PUF parameters (size, decay time, initialization vectors, temperature, etc.) and all combinations thereof is a very time-consuming process and thus only allows for a limited number of measurements.

PIC16F1825: The PIC16F1825 [Mic15] is an 8 bit low-power microprocessor developed by Microchip. The microprocessor holds 1 kB of static RAM, 8 kB of flash memory and 256 B of Electrically Erasable Programmable Read-only Memory (EEPROM). The MCU was specifically designed to be used in medical devices, in automotive scenarios or as part of home appliances.

ATmega328P: The ATmega328P [Atm16] is a low-power 8 bit processor produced by Atmel as part of the megaAVR series. It has 2 kB of SRAM, 32 kB of flash and 1 kB EEPROM on board. The ATmega328P was designed for applications in highly competitive markets where production costs have to be very low.

MSP430F5308: The MSP430F5308 [Tex17a] is a low-energy 16 bit processor produced by Texas Instruments. The device holds 6 kB of static RAM and 16 kB of flash memory. It is optimized for low current drain and thus is used in battery-backed devices for energy constrained applications which require a long service life.

STM32x: The STM32F100Rx series [STM17] is a 32 bit microprocessor equipped with an ARM Cortex-M3 CPU. We investigated two models of this type: i) The STM32VL-Discovery evaluation board, which contains a STM32F100RB microprocessor and ii) the stand-alone STM32F100R8 microprocessor.

Both versions hold 8 kB and 64 kB (STM32F100R8) SRAM and 128 kB flash memory (STM32-F100RB). The evaluation board provides several peripherals and a debugging interface. The STM32F100 series was developed with focus on industrial-control applications. The embedded Cortex-M3 is a low-power MCU used in power sensitive and high performance applications.

LM4F120H5QR: The LM4F120H5QR is a 32 bit ARM Cortex-M4F MCU with 32 kB SRAM, 256 kB flash memory and several programmable interfaces. For our measurements we used the Texas Instruments EK-LM4F120XL development board [Tex14], which is based on this microprocessor. The microprocessor is developed for a variety of industrial applications ranging from electronic point-of-sale machines, and network appliances to factory automation. The Cortex-M4F is conceptually equivalent to the Cortex-M3 but additionally supports instructions for digital signal processing and features a floating-point unit.

PandaBoard: The PandaBoard is 32 bit OMAP4-based SoC platform [Tex17b]. It integrates two ARM Cortex-A9 processors as well as two Cortex-M3 co-processors for signal processing. The PandaBoard is based on a Texas Instruments on an OMAP4460 running at 1.2 GHz. The platform is equipped with 1 GB of external DDR2 DRAM from ELPIDA, implemented as Package-on-Package (PoP), which operates at 1.2 V. The dual-core Cortex-A9 is a high-performance application processor for low-power or cost sensitive devices.

Intel Galileo: The Intel Galileo [Int16a] is a 32 bit SoC equipped with an Intel Quark X1000 application processor, operating at 400 MHz. The platform implements 512 kB SRAM two 128 MB DDR3 DRAM modules from Micron, operating at 1.5 V and 11 kB EEPROM. The two physical DRAM modules are accessed in parallel and located on the same Printed Circuit Board (PCB) as the processor.

4.3. Firmware Modification

In order to extract PUF instances from the MCUs under test for evaluation, we modified the firmware running on the devices. This approach of measuring the intrinsic PUFs proved to be an efficient process regarding the implementation overhead and the process of obtaining repeated PUF measurements. At the same time this approach is in line with the requirement to create software-only means to extract PUF instances (cf. Chapter 1.1).

For all three PUF types, we modified the boot loader part of the firmware image to extract PUF measurements. The rationale behind this approach was initially motivated by the fact that evaluation started with the SRAM PUFs, whose successful extraction requires this very process. Thus, in order to retrieve the largest possible fraction of the SRAM start-up values, they must be read before any interfering process writes to the SRAM, i.e., at the very start of the MCU.

The same approach turned out to be equally efficient for both DRAM-based PUF types. Note however, that one of the main advantages of both DRAM PUFs are due to their capability to be accessed at runtime. For this purpose, novel mechanisms were developed that allow for interfacing the DRAM PUF at runtime, using a Linux kernel module, as presented in [Xio+16].

4.3.1. SRAM-based PUFs

In order to extract the raw SRAM start-up values we developed a custom firmware, which performs the following steps on power-up:

1. Initialize the serial port using Univeral Asynchronous Receiver Transmitter (UART).
2. Loop over the SRAM region, read every SRAM byte and transmit over UART.
3. Ground the device and idle it for 10 s to 15 s to avoid any remanence effects [Rah+12] of the SRAM that could disturb subsequent start-up values.

While designing the firmware we had to take into account the important constraint that, if possible, no SRAM should be used while executing the commands to complete the steps described above. The reason for this is that any write commands to SRAM remove parts of the start-up values. This requirement is easily met for most MCUs which possess several working registers that can be leveraged to store parts of the data used by the firmware modification, i.e. the address of the current SRAM byte to be read. However, some MCUs, such as those of the Microchip PIC16 family, only have a single working register and we therefore need to store variables in unused configuration registers to avoid writes to SRAM.

4.3.2. Decay-based DRAM PUFs

The firmware is the first code to be executed upon device start. During the DRAM initialization phase the firmware itself does not require use of the DRAM, as it is executed from on-chip SRAM. This makes it ideal for gathering PUF measurements.

In the case of the Galileo platform, we modified the Quark EDKII firmware. The code that measures the PUF was inserted just before DRAM refresh, comprising the following steps:

1. Writing the initial value (`iv`) to the specific logical PUF (as defined by `addr` and `size`),
2. disabling the DRAM self-refresh mechanism,
3. waiting for the decay time `t` to elapse,
4. enabling the DRAM self-refresh and
5. reading the logical PUF response and transmitting it via UART.

After the PUF response is retrieved, normal firmware execution and eventual boot of the Operating System (OS) can resume. The firmware patch consists of about 60 lines of `C` code, with the majority of the code implementing initialization of the PUF parameters and accessing the PUF memory region. On the PandaBoard, the implementation is similar: the DRAM region corresponding to the PUF is initialized, the auto-refresh of the memory controller is disabled, and after decay time `t`, the memory content is sent via UART to a workstation. Our firmware patch for the PandaBoard consists of about 50 lines of `C` code.

4.3.3. Rowhammer PUFs

We implemented the Rowhammer PUF on the PandaBoard [Tex17b]. Again, our implementation is purely in software, so no hardware changes are required. The Rowhammer PUF is implemented in the u-boot boot loader. Since the DRAM is idle during u-boot runtime, queries to the Rowhammer PUF can be conducted without affecting other functions of the platform. In u-boot, one can control the DRAM refresh cycle. Further, one can access physical DRAM addresses without caching[3]. The reference manuals provide the physical address mapping of the DRAM. We allocate hammer rows and PUF rows and make them adjacent, as shown in Figure 3.7. To perform Rowhammer, the hammer rows need to be activated repeatedly for a certain time. In our implementation this is achieved by a read operation to the first word of each hammer row. The implementation consists of approximately 200 lines of `C` code. Note that it is also possible to access the Rowhammer PUF in a kernel module to achieve runtime access. Like in u-boot, the DRAM refresh can be disabled from kernel space. Moreover, caching can be disabled if the platform does not support the `CLFLUSH` instruction.

4.4. Evaluation Metrics

As introduced in Chapter 3.1.1, PUFs can be characterized by a number of properties. For identification purposes a PUF instance should enable a robust repeated identification of single devices (*robustness*) and generate a unique pattern among a pool of similar devices (*uniqueness*). Furthermore, we evaluate the *entropy* of the PUF measurements, as in this work secret key generation is the main use case.

This set of properties seems to be the most reasonable, as it omits any physical phenomena that are specific to only a subset of the PUF types (i.e., stable state bias of SRAM PUFs vs. charge decay of DRAM PUFs) and thus makes a comparison of evaluation results practical.

However, evaluating PUF types that rely on different memory technologies (SRAM vs. DRAM) leads to differences that cannot be fully abstracted by considering a common set of PUF properties. Instead, we have to rely on different metrics regarding the properties to properly capture the characteristics of the different PUF types. In this way the comparison of PUF behavior of both memory technologies is possible. We rely on Hamming distance-based metrics as used in the literature [H+07; CLB11; Sel+11] as the standard measures for SRAM-based PUFs. In contrast, we employ metrics that are based on the Jaccard index for the evaluation of both DRAM-based PUFs in order to deal with the inherent high bias of the two DRAM-based PUFs as shown in the following section. We further employ *fractional* metrics to make a comparison of results across different device types easier. Fractional metrics are normalized by the considered PUF size. In particular the metric m_{frac} is divided by the number of input PUF bits N:

$$m_{\mathsf{frac}} = \frac{m}{N}. \tag{4.1}$$

3 The PandaBoard implements an ARM processor that does not provide the `CLFLUSH` instruction, so caching is disabled during Rowhammer PUF access.

Using fractional metrics provides more meaningful results, in face of highly varying memory sizes of the different device types. Throughout this thesis we will use fractional results and their percental representation synonymously, i.e., $m_{\mathsf{frac}} = 0.814 \mathrel{\widehat{=}} 81.4\,\%$.

4.4.1. Metrics Based on Hamming Distances

A PUF measurement X obtained from an SRAM-based PUF is regarded as a bit array of zeros and ones. Standard metrics proposed in the literature to evaluate SRAM-based PUFs are based on the Hamming distance **HD**(X_i,X_j) computed between pairs of measurements (X_i,X_j).

The *Hamming weight* (**HW**) of individual measurements from the same device indicates whether the start-up values are biased towards the value zero or one. This metric gives a first indication regarding the randomness of the start-up values. The ideal Hamming weight follows a Gaussian distribution with a mean of 50 %, indicating no bias in the start-up values.

The *intra-Hamming distance* ($\mathbf{HD_{intra}}$) reflects the stability of repeated PUF measurements for a single device, when queried by a fixed challenge. Robustness of the start-up values is required to reliably identify a given device and subsequently reconstruct the corresponding cryptographic key. $\mathbf{HD_{intra}}$ is a normalized count of bits which differ between subsequent PUF measurements and thus is a rational number between zero and one. An optimal value for the intra-Hamming distance is close to zero. However, most PUF measurements exhibit a certain amount of noise. In case of the SRAM-based PUF this is due to cells that have a rather symmetric layout of the transistors that reflect in flipping start-up values over multiple trials.

The *inter-Hamming distance* ($\mathbf{HD_{inter}}$) expresses whether the start-up values at the same bit positions are diverse enough across different devices of the same type. This metric indicates whether start-up values can be used for identification without enabling adversaries to predict a measurement for a second device on the basis of measurements of the first device. In the optimal case, the inter-Hamming distance follows a Gaussian distribution with a mean of 50 %. If this is true, then the start-up values are most likely independent. Devices with an optimal value of 50 % exhibit a maximum distinguishability regarding their PUF responses.

4.4.2. Metrics Based on the Jaccard Index

The characteristics of PUFs that are based on the DRAM are different compared to SRAM PUFs. In case of the SRAM-based PUF, ideally the number of zero and one bits in the PUF measurements is equal and hence no bias towards one of the values exists. However, measurements obtained from the two DRAM-based PUF types only show a small number bit flips. This is caused by the fact that the majority of DRAM cells does not decay within typical timescales of PUF challenges. Hence, there is a strong bias towards the initialization value. Subsequently, standard metrics commonly used to evaluate memory-based PUFs, i.e., fractional Hamming distances, are not suitable for the DRAM case due to the high bias. This is particularly noticeable when evaluating uniqueness. In SRAM PUFs the fractional Hamming distance between the startup arrays of two different PUFs is large, whereas for DRAM PUFs the distance is small, even if PUFs are highly unique. Thus, we propose metrics based on the Jaccard index to evaluate DRAM PUF characteristics that ignore the “uninteresting” majority of cells, i.e., those cells that did not decay.

4.4.3. Metrics Used for Entropy Estimation

Since the main scenario of the investigated intrinsic PUF instances is the extraction of a device-unique cryptographic key, an important requirement of the PUF measurements is to exhibit enough randomness, which is reflected in the entropy of the derived key. A typical recommendation in symmetric key scenarios is to use Advanced Encryption Standard (AES) with a minimum key size of 128 bit [Bar16].

In order to evaluate the entropy that is inherent to PUF measurements, we consider the *inter* entropy, i.e., the estimated entropy across measurements obtained from different devices. Obviously, it is desired to observe a high inter entropy, so that a derived symmetric key with the recommended key size can be constructed from a PUF measurement that is as small as possible. In contrast, to generate random numbers, one would evaluate the intra entropy obtained from a single PUF instance, which is comparably small [Van+13].

In order to estimate the inter-device entropy, we employ different metrics, depending on the PUF type and hence on the underlying memory technology, namely:

- min-Entropy for SRAM-based PUFs to provide a lower bound on the entropy,
- the Context Tree Weighting (CTW) method, also applied to SRAM-based PUFs, in order to provide a less pessimistic entropy estimation that is not based on independence assumptions and the
- Shannon Entropy for DRAM-based PUFs, in order to consider the inherent bias of the PUF responses.

The Shannon entropy $\mathbf{H}(X)$ provides a measure regarding the amount of information that can be extracted from observing a variable X. In other words, the Shannon entropy provides an estimate on the uncertainty of a given variable and is usually expressed in bits.

The min-entropy $\mathbf{H}_{\infty}(X)$ is the standard metric used in the literature to quantify entropy of SRAM PUF measurements [Lee+12]. This method is based on the NIST specification [BK12] that defines min-entropy as the worst-case metric of disorder of a random variable. The min-entropy is based on the assumption that start-up bits are independent from each other. Although this assumption is commonly accepted in the literature with respect to SRAM PUF instances [VSL13; CLB11; SSL12] exceptions are possible as can be seen in the following section. While the Shannon entropy provides a weighted average of the information content of a variable X, min-entropy is the greatest lower bound, only considering the most likely outcomes of a random process. Hence, relying on the approach of min-entropy, one is guaranteed to always exhibit its result as the minimum entropy of a random process. Hence, it holds that $\mathbf{H}_{\infty}(X) \leq \mathbf{H}(X)$.

In order to provide a less pessimistic approach to estimate the entropy of SRAM PUF measurements and at the same time to detach the evaluation from the assumption that start-up bits are independent, we estimated entropy based on the CTW. This method is an optimal compression algorithm that provides an estimate on the upper bound entropy in the PUF responses. Using CTW to estimate the creation entropy of measurements of different PUFs is a well accepted approach [Ign+06; Van+10; CLB11; SL12]. Given a measurement X_o of size l_o and its CTW-compressed version X_c with respective size l_c, the compression ratio can be computed as:

$$r_{CTW} = \frac{l_c}{l_o}. \tag{4.2}$$

The intuition behind this approach is due to the fact that if no compression is possible, all bits must be random. Hence, by using CTW one estimates entropy on the basis of the length of the compressed measurement. In particular, a compression ratio below 100 % indicates that some bits of the start-up values exhibit a linear dependency. However, the compression could also result from the biases of the cells, since the CTW is only a weak indicator for correlation [VSL13].

4.5. Evaluation Results

In the following section, we present the evaluation results for the three PUF types investigated. For each PUF type, we evaluate uniqueness, robustness and entropy. Furthermore, we present evaluation results obtained under different ambient conditions, mainly extreme temperatures and measurements obtained after long-term periods of usage. Note that an analysis of the PUF types under non-nominal voltage conditions is usually not possible as most COTS devices usually have no interface that allows for changes to the input voltage.

4.5.1. SRAM-based PUF Results

In this section we present the results of our analysis of the SRAM start-up values obtained from different device types. All devices have been tested at room temperature of approximately 20 °C. For all devices the first measurement at room temperature has been used for enrollment. All other measurements are compared to the enrollment measurement. The odd numbers of measurements are due to the fact that during the experiment some erroneous measurements were obtained, which have been removed from the data set before analysis.

For specific devices, we also conducted measurements in a climate chamber at different (extreme) temperatures: −30 °C, 20 °C and 85 °C to 90 °C. However, not for all device types temperature test results are provided. The STM32F100RB returned false temperature measurements with misaligned bytes at random positions that made evaluation impossible as correct bit-wise alignment of repeated measurements is crucial for analysis. This was most probably caused by the USB-to-UART converter, needed to transfer measurements from the evaluation platform to the workstation. The converter introduced or even suppressed bytes randomly at extreme temperatures. Also, the PandaBoard was not exposed to extreme temperatures as it features an LC display and other hardware elements which would be destroyed inside the climate chamber.

For each metric presented in Chapter 4.4.1 we present plots showing the averaged characteristics of different device types to allow for comparison between them. The averaged values were obtained by first creating all possible pair-wise combinations of measurements for every device. Subsequently, for each measurement pair the corresponding metric was calculated. In order to compare metrics of different device types, the averaged data was further averaged over the individual devices to get a value that represents the entire device type. In addition, we present worst-case values of uniqueness and robustness metrics, i.e., respective minimum or maximum values, in Table 4.2. Again, those values were obtained by aggregating the respective minimum

Device Type	N_{bits}	**HW** (min ; max)			$\mathbf{HD}_{\mathbf{intra}}$ (max.)			$\mathbf{HD}_{\mathbf{inter}}$ (avg.)
		−30 °C	20 °C	≈ 85 °C	−30 °C	20 °C	≈ 85 °C	20 °C
PIC16F721	8192	0.432 ; 0.503	0.457 ; 0.515	0.483 ; 0.515	0.106	0.020	0.079	0.213
ATMega328p	16384	0.701 ; 0.762	0.644 ; 0.686	0.584 ; 0.628	0.123	0.026	0.071	0.443
MSP430F5308	49152	0.571 ; 0.676	0.600 ; 0.665	0.586 ; 0.664	0.116	0.046	0.069	0.463
STM32F100R8	65536	0.491 ; 0.499	0.492 ; 0.498	0.492 ; 0.497	0.080	0.057	0.106	0.477
STM32F100RB	65536	——	0.494 ; 0.499	——	——	0.093	——	0.472
LM4F120H5QR	262144	0.442 ; 0.589	0.433 ; 0.537	0.425 ; 0.513	0.053	0.053	0.065	0.493
PandaBoard (ES)	12288	——	0.452 ; 0.509	——	——	0.043	——	0.499

Table 4.2.: Evaluation results of bias, uniqueness and robustness for SRAM-based PUF instances extracted from various device types under different ambient temperatures.

or maximum values for single devices and further calculating the minimum or maximum values respectively for the entire device type.

Hamming weight

In the following paragraph, we present the results of the Hamming weight (**HW**) characteristics of the measured devices. Figure 4.1 compares the average Hamming weight per device type. For each individual MCU the corresponding whisker boxes display the **HW** results. Detailed numbers of the average Hamming weights can be seen in Table 4.2. The values shown are the minimum and maximum values from the averaged **HW** results of each individual MCU instance.

The results for the PUF responses of the STM32F100R8, the STM32F100RB and the PandaBoard exhibit properties that are close to the desired distribution, indicating that the start-up values contain almost the same proportion of zeros and ones. Especially, the **HW** values of the STM32-F100R8 gather nicely close to the optimal value of 50 % with almost no differences between the

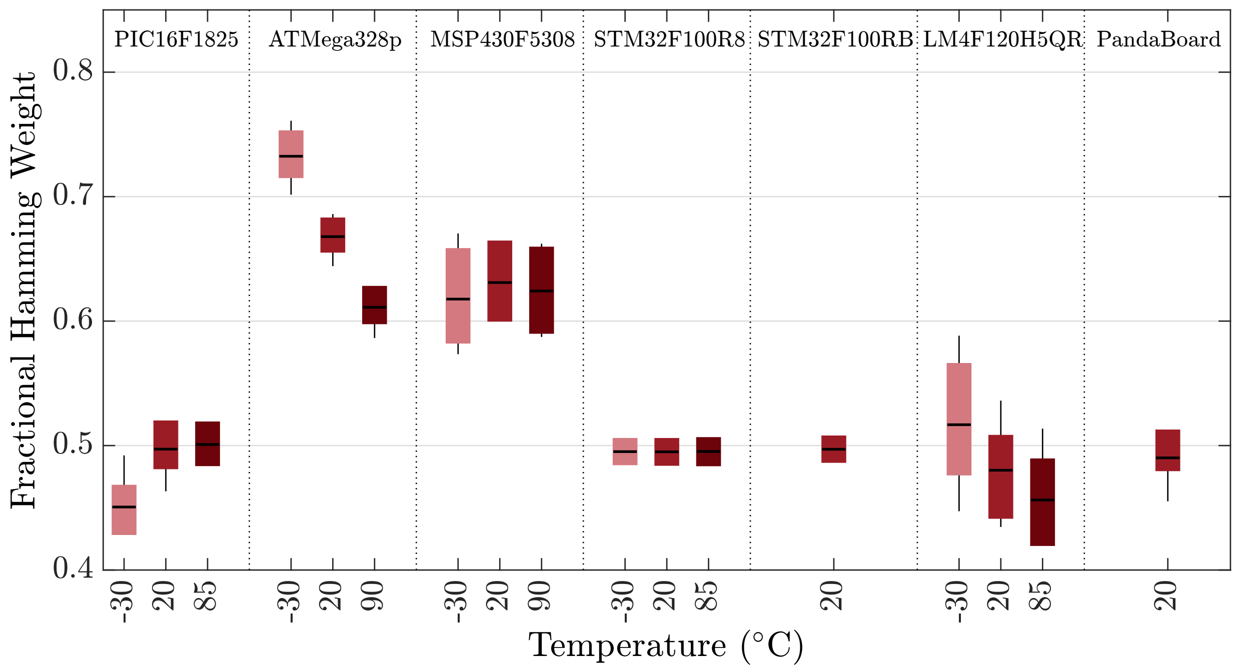


Figure 4.1.: Hamming weight (**HW**) values of SRAM start-up measurements obtained from device types at different ambient temperatures (optimum: 0.5).

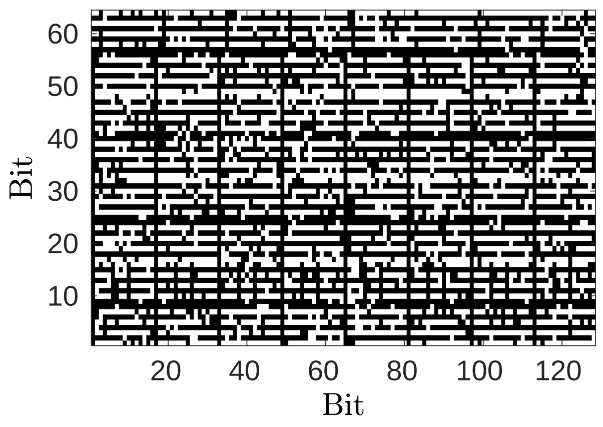


Figure 4.2.: Repeating patterns in the 8 kB SRAM start-up values from an PIC16F721 MCU.

different temperature measurements, showing almost perfect HW characteristics. The **HW** for the LM4F120H5QR shows reasonable **HW** results with a worst-case Hamming weight of 43.29 % at 85 °C. This reveals that there is a certain correlation among the start-up values of similar devices, which decreases randomness. Hence, the required portion of SRAM to derive a unique fingerprint and eventually a cryptographic key with full entropy increases [Van+10]. In particular, due to the slight bias towards zero, the secrecy rate decreases, which in turn requires a higher amount of SRAM start-up bits in the privacy amplification phase of the Fuzzy Extractor. For details regarding the relation between potential correlation in start-up values, the secrecy rate metric and the resulting number of SRAM bits required by a Fuzzy Extractor construction, we refer to [Ign+06]. The Hamming weight of the start-up values generated by the ATmega328P as well as the MSP430F5308 are significantly higher than 50 % at any of the tested temperatures. This indicates a higher portion of ones than zeros in the PUF responses. The PIC16F1825's fractional Hamming weight is close to 50 %, which at first glance represents desired characteristics. However, a visual examination of the PUF responses reveal a repeating pattern (cf. Figure 4.2), which will be discussed later in this section.

The temperature dependency of the Hamming weight values differ among the tested device types. Some types, including the MSP430F5308 and the STM32F100RB, are very stable also at extreme ambient temperatures. The evaluation results of other types, including the ATMega328p and the LM4F120H5QR, suggest that at low temperatures more SRAM cells initialize to one, whereas at high temperatures more zero bits constitute the start-up patterns. Lastly, for the PIC16F1825 devices, merely low temperatures seem to influence, i.e., to decrease the Hamming weight values, while at higher temperatures no significant changes are observable. Given this heterogeneous behavior, we refer temperature characteristics to the low-level physical structures of the implemented SRAM cells and their interconnection via word- and bit-lines. Unfortunately, such low-level information is not available, especially in the case of COTS devices, as it is regarded as intellectual property of the vendor. Hence, no further reliable statements with respect to the relationship between temperature conditions and PUF characteristics can be made.

The **HW** numbers for all devices suggest that their SRAM start-up values are suitable inputs for commonly known Fuzzy Extractors, even for those devices with a Hamming weight that cannot be regarded as optimal, due to bias. These inferior characteristics will lead to an increased PUF response size as input to a Fuzzy Extractor algorithm to reliably extract a unique identifier.

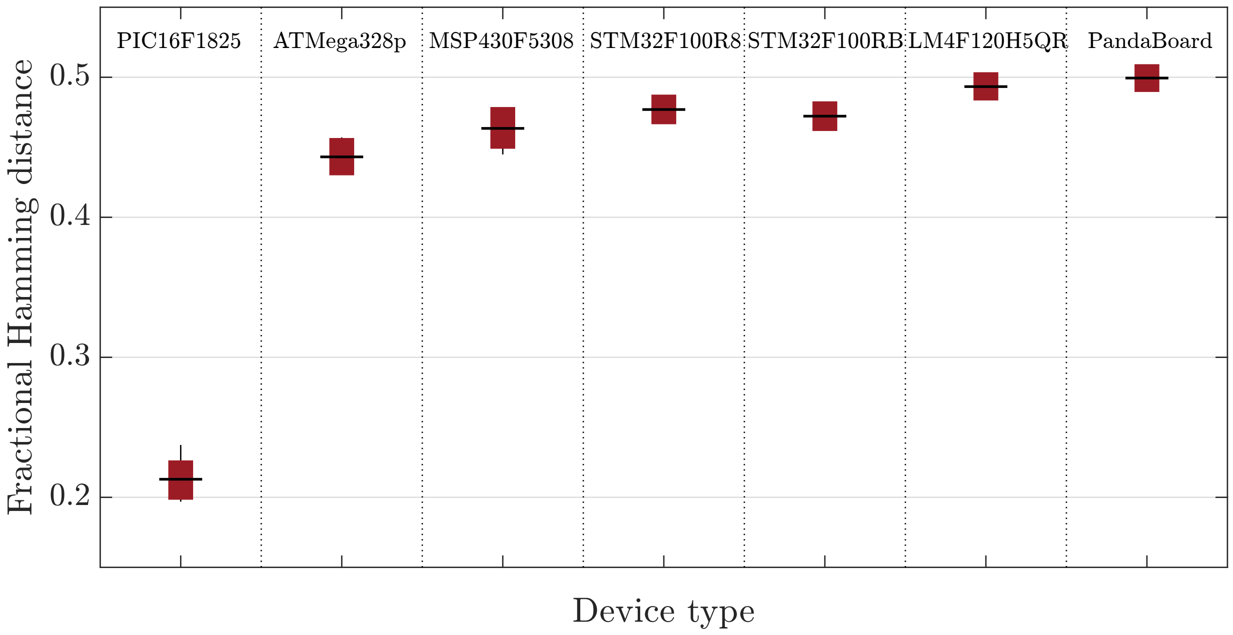


Figure 4.3.: Uniqueness values ($\mathbf{HD_{inter}}$) of SRAM start-up measurements obtained from device types at different ambient temperatures (optimum: 0.5).

Uniqueness

Figure 4.3 compares the average inter-Hamming distances ($\mathbf{HD_{inter}}$) to each other. An overview of averaged $\mathbf{HD_{inter}}$ values is given in Table 4.2. Whilst some device types exhibit near-perfect between-class Hamming distances with a mean of 50 % (e.g., the PandaBoard), this is not the case for all measured devices. The PIC16F1825 is especially deficient in this regard, having a maximum measured $\mathbf{HD_{inter}}$ of only 24.54 %, making it unsuitable for unique identification. The $\mathbf{HD_{inter}}$ of the PIC16F1825 is much lower than required for a PUF implementation. PUF measurements of this device type fit a Gaussian distribution with a mean value of 21.29 %. This low mean value indicates that for the PIC16F1825 there is a very high correlation between the PUF responses from different devices. This assumption becomes even more apparent when considering the min-entropy data of the PIC16F1825. Especially the repetitive pattern in the start-up values (cf. Figure 4.2) causes PUF responses obtained from different devices to share more similar start-up bits compared to the other devices. This makes them unsuitable for usage as part of cryptographic algorithms.

The PUF responses of the other device types exhibit an inter-Hamming distance which is close to the optimal 50 %. They can be uniquely identified as the $\mathbf{HD_{inter}}$ between different devices is much higher than the noise measured for each individual device (i.e., the $\mathbf{HD_{intra}}$). Especially the $\mathbf{HD_{inter}}$ of the LM4F120H5QR and the PandaBoard are remarkable, since they are close to the optimum. These numbers guarantee to provide a unique fingerprint for individual devices among a pool of devices of the same type.

Note that the measurements at extreme temperatures are omitted as the $\mathbf{HD_{inter}}$ values are computed on enrollment data. In the scenario of extracting a device fingerprint the keys are derived at room temperature. During reconstruction, the keys are merely recreated. Thus, the uniqueness of the key – and subsequently the SRAM start-up values – must be guaranteed at the time of enrollment. Given that enrollment data was measured at room temperature it is not useful to compare it to measurements at other temperatures.

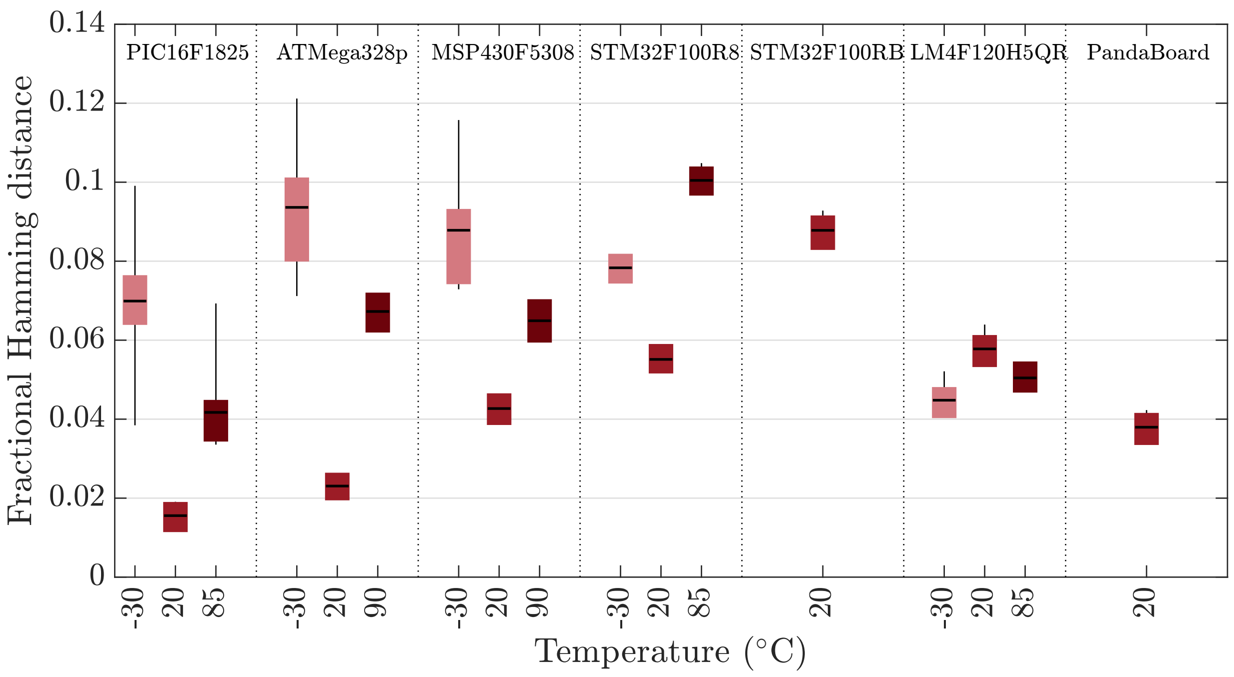


Figure 4.4.: Robustness values ($\mathbf{HD_{intra}}$) of SRAM start-up measurements obtained from device types at different ambient temperatures (theoretical optimum: 0).

Robustness

The results of the averaged intra-Hamming distance ($\mathbf{HD_{intra}}$) of the analyzed MCUs can be seen in Figure 4.4. Since the $\mathbf{HD_{intra}}$ refers to the noise exhibited in the SRAM start-up values of a given device across multiple activation operations, it is desired to have $\mathbf{HD_{intra}}$ values as low as possible. The existing noise needs to be eliminated using an error correcting scheme. For this purpose, commonly a Fuzzy Extractor is used. A reference Fuzzy Extractor design presented in [Bös+08] can correct up to 15 % noise. Thus, besides the requirement of the $\mathbf{HD_{intra}}$ to be as low as possible, it should not exceed this threshold. Detailed intra-Hamming distance results can be found in Table 4.2. We only list the recorded maximum values here, since that is what matters when selecting appropriate parameters for error correction algorithms. Again, the listed values are the maximum results from the averaged $\mathbf{HD_{intra}}$ values of each measured MCU instance.

The maximum $\mathbf{HD_{intra}}$ value at room temperature for the PIC16F1825 and the ATmega828P is below 3 %, indicating nearly perfect characteristics so that noise can be easily corrected using commonly known Fuzzy Extractors. The MSP430F5308, the STM32F100R8, the LM4F120H5QR and the PandaBoard show good robustness properties at room temperature with noise levels from 4 % to 6 %. In contrast, the STM32F100RB exhibits reasonable robustness characteristics with a noise level of 9 % already at room temperature. The temperature measurements of the ATmega328P, PIC16F1825 and the MSP430F5308 exhibit a significant increase of the $\mathbf{HD_{intra}}$ values at low temperatures for all tested devices, which indicates that deviation from the nominal operational (room) temperature, leads to less stable PUF measurements.

Gathered robustness values for all the tested devices are well below the critical threshold of 15 % so that the inherent noise can be corrected using a common Fuzzy Extractor. However, in consideration of the relative differences between room temperature and extreme temperature of some device types, it is obvious that MCUs need to be tested thoroughly with regard to ambient conditions if they are supposed to be used in security-related applications.

Device Type	min-entropy (min.)			CTW		
	−30 °C	20 °C	≈ 85 °C	−30 °C	20 °C	≈ 85 °C
PIC16F721	0.215	0.191	0.174	0.536	0.615	0.591
ATMega328p	0.459	0.579	0.668	0.843	0.922	0.969
MSP430F5308	0.432	0.629	0.640	0.951	0.948	0.956
STM32F100R8	0.613	0.637	0.684	0.993	0.998	1.000
STM32F100RB	——	0.651	——	——	0.871	——
LM4F120H5QR	0.684	0.720	0.685	0.985	0.990	0.987
PandaBoard	——	0.633	——	——	1.000	——

Table 4.3.: Entropy results of SRAM-based PUF measurements extracted from various device types under different ambient temperatures.

Entropy

Table 4.3 shows the inter-device min-entropy results for the tested devices. We list the minimum measured min-entropy, since this is the value to take into account when determining how much SRAM is required to derive a cryptographic key. In general, all the tested device types exhibit less min-entropy than SRAM-based PUF instances evaluated in previous works, that usually show min entropy values of 0.7 to 0.9 [Kat+12; Koe+14a]. Presumably this is due to the type of SRAM we investigated, which is on-chip, i.e., intrinsically part of the embedding MCU. In contrast, previous min-entropy results of SRAM-based PUFs were obtained from either Application-Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) implementations as well as off-chip modules. Furthermore, the min-entropy results show that there are significant differences between individual device types. Whilst most of the devices show good results when it comes to deriving a unique key from the SRAM start-up patterns, the PIC16F1825 MCUs are not suitable for this purpose. As explained during the uniqueness analysis, the PUF measurements of this device type exhibit repetitive structures. In particular, the bits of every alternating byte have a common preference to start-up either as a zero or one. This pattern is present in every PIC16F1825 device we measured. It is also reflected in the low min-entropy values of this device type. One possible reason for such patterns are (analog) components connected to the power supply, which distort the ramp-up on the SRAM. Microchip does not provide information about their silicon implementation, which makes it impossible for us to investigate what is happening inside the devices.

It is noteworthy that the min-entropy results for almost all tested device types are relatively stable under different thermal conditions. Nevertheless, the min-entropy results suggest that lower temperatures slightly decrease the min-entropy results, while increased temperatures lead to higher values. This suggests that at lower temperatures, the start-up values extracted from device instances of the same device type observe a higher similarity. One possible explanation is that at lower temperatures the influence of the physical layout of the SRAM cells becomes more significant on the start-up values. In particular, for some cells it seems that the mismatch between the threshold voltages of the corresponding cross-couples inverters increases at lower temperatures. This increased mismatch seems to align with the physical layout that underlies all instances of a given device type. The opposite behavior might explain increased min-entropy values at higher temperatures. In general, all the device types except for the PIC16F1825 show fairly good entropy results, sufficient to extract enough entropy for a cryptographic key. For example, the STM32F100R8 devices exhibit a minimum amount of 61.3 % min-entropy in every thermal condition. In order to extract a robust 128 bit key one needs approximately 27 B of SRAM start-up values (not considering the entropy required to compensate leakage due to Helper Data exposure), which is a comparably small fraction of the total available 8 kBSRAM.

Table 4.3 further lists the results of the CTW compression tests. In particular, single PUF measurements from different devices were concatenated and the CTW compression algorithm [WST95] was applied. Subsequently, the compression ratio r_{CTW} was calculated according to Equation (4.2). Both STM-based MCUs, the LM4F120H5QR and the PandaBoard exhibit almost maximum r_{CTW} values, indicating high independence between the SRAM start-up bits. While the MSP430F5308 and the ATMega328p show reasonable characteristics, which are mostly due to the stronger bias towards 1, the PIC16F1825 again features poor compression results. Given that there is no strong bias observable in the PIC16F1825 start-up values, we attribute this to the byte-wise repeating structures in the SRAM measurements (cf. Figure 4.2).

4.5.2. Decay-Based DRAM PUF Results

We measured DRAM PUF instances on the Intel Galileo and PandaBoard, as described in Chapter 3.3.1. We performed measurements using four different PandaBoards and five Intel Galileo devices. Furthermore, given the large amount of memory present, we measured two logical PUFs on each device, resulting in eight different logical PUFs for the PandaBoard as well as ten logical PUFs for the Intel Galileo. Each logical PUF was measured at different decay times, with 50 measurements each. We used two groups of configurations. One group consists of decay times $\mathcal{T}_2 = \{120\,\text{s}, 180\,\text{s}, 240\,\text{s}, 300\,\text{s}, 360\,\text{s}\}$ and a smaller PUF size of 32 kB. The other group uses decay times $\mathcal{T}_1 = \{10\,\text{s}, 20\,\text{s}, 30\,\text{s}, 40\,\text{s}, 50\,\text{s}, 60\,\text{s}\}$, which are up to six times faster and further covers a larger PUF size of 16 MB. Based on these measurements we evaluated robustness, uniqueness, randomness, time and temperature dependency, as well as stability of the DRAM PUFs. In order to present a realistic scenario, we tested our devices under conditions that naturally vary over time, in order to resemble ambient properties during real-world usage. The evaluation results are listed in Table 4.4.

$\mathcal{T}_1 = \{10\,s, 20\,s, 30\,s, 40\,s, 50\,s, 60\,s\}$, `size` $= 16\,MB$					
decay times $\mathcal{T}_1$	device type	min. $\mathbf{J}_{intra}$	max. $\mathbf{J}_{inter}$	$\mathbf{H}$ (bits)	avg. number decayed cells
10 s	PandaBoard	6.870×10^{-1}	0.000	7.062×10^{3}	5.250×10^{2}
	IntelGalileo	3.850×10^{-1}	0.000	3.810×10^{2}	2.300×10^{1}
20 s	PandaBoard	7.120×10^{-1}	3.000×10^{-4}	6.741×10^{4}	6.132×10^{3}
	IntelGalileo	4.750×10^{-1}	0.000	3.837×10^{3}	2.720×10^{2}
30 s	PandaBoard	7.260×10^{-1}	1.000×10^{-3}	2.294×10^{5}	2.380×10^{4}
	IntelGalileo	4.650×10^{-1}	1.000×10^{-3}	1.508×10^{4}	1.194×10^{3}
40 s	PandaBoard	7.380×10^{-1}	1.000×10^{-3}	5.099×10^{5}	5.835×10^{4}
	IntelGalileo	5.140×10^{-1}	4.000×10^{-4}	4.266×10^{4}	3.711×10^{3}
50 s	PandaBoard	7.620×10^{-1}	2.000×10^{-3}	8.973×10^{5}	1.108×10^{5}
	IntelGalileo	5.500×10^{-1}	2.000×10^{-4}	8.658×10^{4}	8.078×10^{3}
60 s	PandaBoard	7.690×10^{-1}	3.000×10^{-3}	1.374×10^{6}	1.805×10^{5}
	IntelGalileo	5.880×10^{-1}	4.000×10^{-4}	1.478×10^{5}	1.459×10^{4}
$\mathcal{T}_2 = \{120\,s, 180\,s, 240\,s, 300\,s, 360\,s\}$, `size` $= 32\,kB$					
decay times $\mathcal{T}_2$	device type	min. $\mathbf{J}_{intra}$	max. $\mathbf{J}_{inter}$	$\mathbf{H}$ (bits)	avg. number decayed cells
120 s	PandaBoard	4.630×10^{-1}	1.000×10^{-2}	1.362×10^{4}	1.069×10^{3}
	IntelGalileo	7.710×10^{-1}	4.000×10^{-3}	3.382×10^{3}	2.450×10^{2}
180 s	PandaBoard	4.380×10^{-1}	1.700×10^{-2}	3.163×10^{4}	2.675×10^{3}
	IntelGalileo	8.360×10^{-1}	4.000×10^{-3}	8.482×10^{3}	6.400×10^{2}
240 s	PandaBoard	4.090×10^{-1}	2.600×10^{-2}	4.736×10^{4}	4.161×10^{3}
	IntelGalileo	6.260×10^{-1}	5.000×10^{-3}	1.381×10^{4}	1.085×10^{3}
300 s	PandaBoard	4.220×10^{-1}	4.100×10^{-2}	5.911×10^{4}	5.307×10^{3}
	IntelGalileo	7.940×10^{-1}	6.000×10^{-3}	1.960×10^{4}	1.588×10^{3}
360 s	PandaBoard	3.480×10^{-1}	3.400×10^{-2}	6.738×10^{4}	6.129×10^{3}
	IntelGalileo	8.280×10^{-1}	7.000×10^{-3}	2.912×10^{4}	2.444×10^{3}

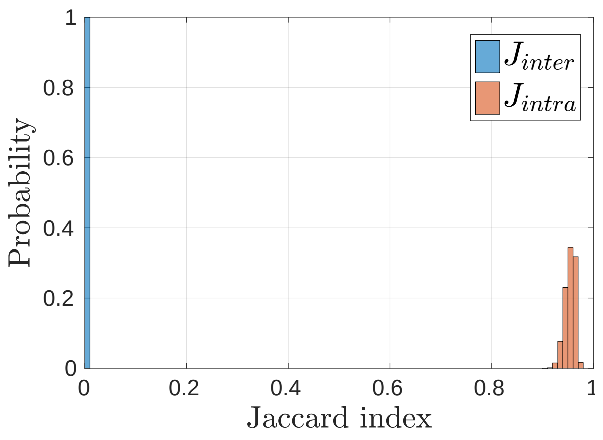
Table 4.4.: Evaluation results for logical decay-based DRAM PUF instances measured at different decay times $\mathcal{T}_1$ and $\mathcal{T}_2$ as well as for different PUF sizes. Top: results for decay times $\mathcal{T}_1 = \{10\,s, 20\,s, 30\,s, 40\,s, 50\,s, 60\,s\}$ and `size` $= 16\,MB$. Bottom: results for decay times $\mathcal{T}_2 = \{120\,s, 180\,s, 240\,s, 300\,s, 360\,s\}$ and `size` $= 32\,kB$.

Uniqueness

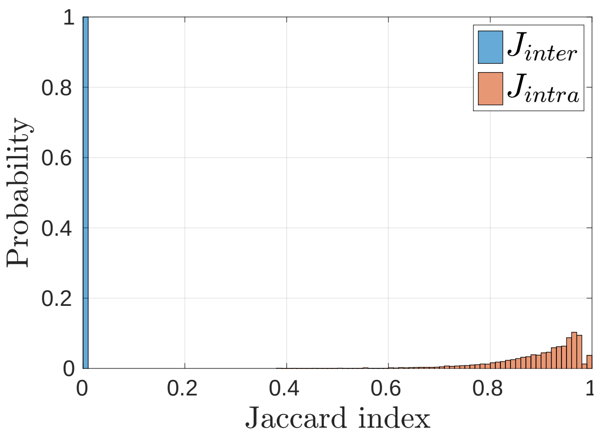
Consider two DRAM PUFs, $\mathbf{PUF}_{id_1}$ and $\mathbf{PUF}_{id_2}$, which are given time $\mathtt{t}$ to decay. Let $s_1(\mathtt{t})$ be the set of addresses of the decayed cells in $\mathbf{PUF}_{id_1}$ and similarly $s_2(\mathtt{t})$ for $\mathbf{PUF}_{id_2}$. Let N be the total number of DRAM cells. The similarity between $\mathbf{PUF}_{id_1}$ and $\mathbf{PUF}_{id_2}$ is expressed as

$$\mathbf{J}_{inter}^{1,2}(\mathtt{t}) = \frac{\mathbf{J}(s_1(\mathtt{t}), s_2(\mathtt{t}))}{N}. \tag{4.3}$$

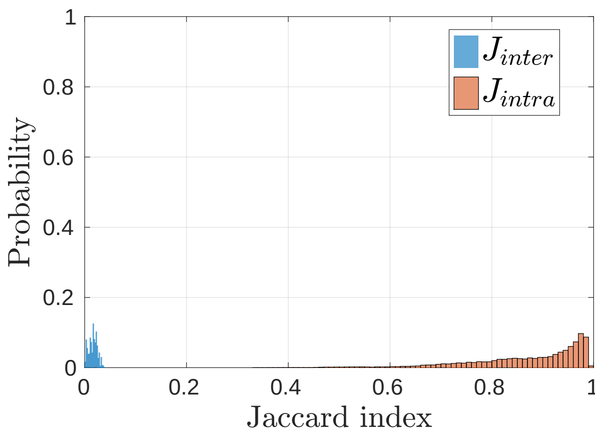
A small value of $\mathbf{J}_{inter}^{1,2}$ indicates high uniqueness. As shown in Table 4.4, our DRAM PUFs exhibit almost perfect behavior, with $\mathbf{J}_{inter}$ values for decay times up to 60 s that do not exceed 0.001 for the Intel Galileo and 0.003 for the PandaBoard. At higher decay times, up to 6 minutes, Intel Galileo exhibits a maximum of $\mathbf{J}_{inter} = 0.007$ at $\mathtt{t} = 360$ s. The PandaBoard shows larger values with a maximum of 0.041 at $\mathtt{t} = 300$ s, which is still close to the optimal value of zero. Those comparably low values at shorter decay times are due to the fact that much less DRAM cells



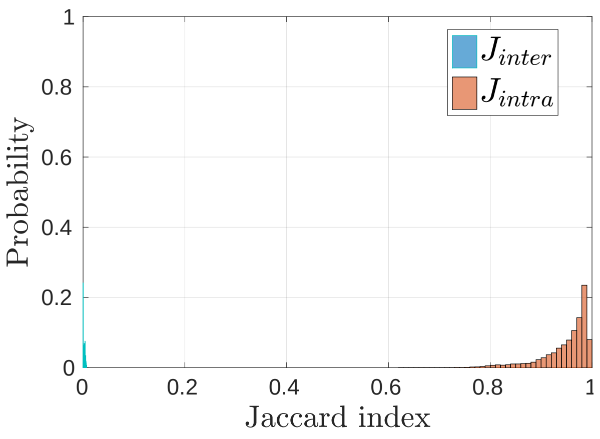
PandaBoard results, obtained at $\mathcal{T}_1$, using $\mathtt{size} = 16$ MB.



Intel Galileo results, obtained at $\mathcal{T}_1$, using $\mathtt{size} = 16$ MB.



PandaBoard results, obtained at $\mathcal{T}_2$, using $\mathtt{size} = 32$ kB.



Intel Galileo results, obtained at $\mathcal{T}_2$, using $\mathtt{size} = 32$ kB.

Figure 4.5.: Histograms of $\mathbf{J}_{intra}$ and $\mathbf{J}_{inter}$ values for the PandaBoard and the Intel Galileo, with $\mathcal{T}_1 = \{10\,\text{s}, 20\,\text{s}, 30\,\text{s}, 40\,\text{s}, 50\,\text{s}, 60\,\text{s}\}$ and $\mathcal{T}_2 = \{120\,\text{s}, 180\,\text{s}, 240\,\text{s}, 300\,\text{s}, 360\,\text{s}\}$ and two different PUF sizes.

have had the chance to decay within these short time periods (cf. Figure 4.6). The values for both configurations suggest that both device types exhibit high uniqueness, with the Intel Galileo showing inherently smaller $\mathbf{J}_{inter}$ values compared to the PandaBoard.

Robustness

Consider again the experiment where a decay-based DRAM PUF is given time t to decay. Let $s(\mathsf{t})$ be the set of addresses of decayed cells in one run of this experiment, and $s'(\mathsf{t})$ in a subsequent run of the experiment on the same PUF_{id}. We characterize the robustness of the PUF as:

$$\mathbf{J}_{intra}(\mathsf{t}) = \frac{\mathbf{J}(s(\mathsf{t}), s'(\mathsf{t}))}{N}. \tag{4.4}$$

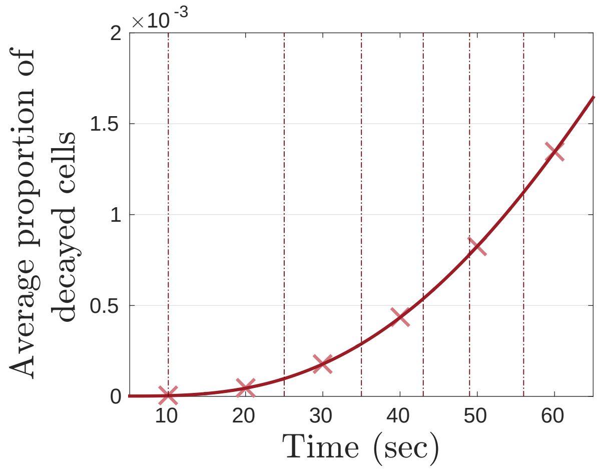
Large values of $\mathbf{J}_{intra}$ indicate high robustness. Figure 4.5 shows the distributions of $\mathbf{J}_{intra}$ and $\mathbf{J}_{inter}$ for different decay times. A wide gap between the two distributions indicates that individual devices can be distinguished perfectly. Note that at shorter decay times we observe minimum $\mathbf{J}_{intra}$ values of 0.385 for the Intel Galileo, whereas for the PandaBoard noise levels are smaller with the minimal $\mathbf{J}_{intra}$ value 0.687 at $\mathsf{t} = 10\,\mathrm{s}$. Using longer decay times, noise levels increase for both device types. While the minimum $\mathbf{J}_{intra}$ value for the Intel Galileo is 0.626 at $\mathsf{t} = 240\,\mathrm{s}$, the PandaBoard exhibits the most noise at $\mathsf{t} = 360\,\mathrm{s}$ with 0.348 (cf. Table 4.4). The differences in the $\mathbf{J}_{intra}$ values among the two groups of configurations reflect variations of ambient conditions (i.e., temperature) over time. Nevertheless, we note that in both cases, the $\mathbf{J}_{intra}$ values are high enough to robustly separate from $\mathbf{J}_{inter}$, in turn allowing the devices to be used successfully for robust identification purposes. We can therefore conclude that our devices constitute robust PUF behavior in a realistic usage scenario, where ambient conditions, such as temperature, are expected to naturally differ.

Entropy

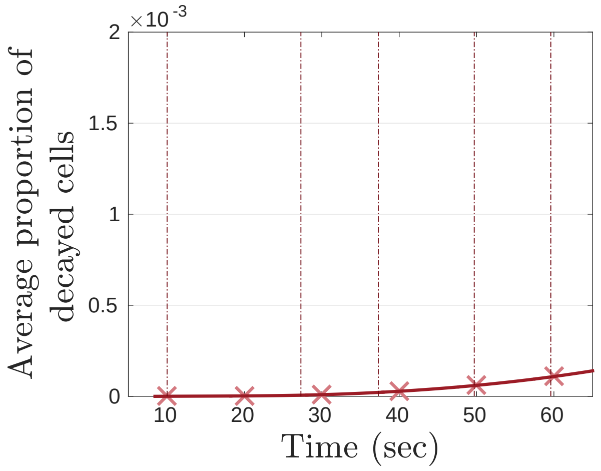
We estimate the entropy of DRAM PUFs in the following manner. We again consider the observed set $s(\mathsf{t})$ of indices of DRAM cells that have decayed by time t. The cardinality of $s(\mathsf{t})$ is denoted as $l_{\mathsf{t}} = |s(\mathsf{t})|$, and N is the total number of DRAM cells. We assume that each DRAM cell independently has a probability $p(\mathsf{t})$ of having a decay time smaller than t (so that it usually decays in time less than t). We estimate $p(\mathsf{t})$ as l_{t}/N. The PUF entropy associated with time t is given by

$$\mathbf{H} = Nh(p(\mathsf{t})) \approx Nh(l_{\mathsf{t}}/N), \tag{4.5}$$

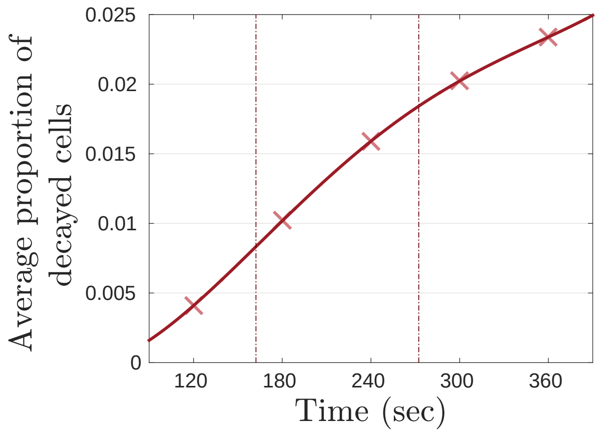
where $h(p) = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p}$ is the binary entropy function. A single observation of $s(\mathsf{t})$ may not be sufficient for determining $p(\mathsf{t})$ because of short-term noise. Thus, we are estimating $p(\mathsf{t})$ by averaging 50 measurements of l_{t}. Table 4.4 lists the entropy $\mathbf{H}(X)$ as bits per measured logical PUF (i.e., 16 MB and 32 kB). We observe that the entropy is significantly higher on the PandaBoard, correlating with the higher number of bit flips of this device type. This is most likely due to the different technologies used to implement DRAM cells. In particular, the results show that the minimum entropy of the PandaBoard can be up to one order of magnitude larger compared to the Intel Galileo: For example at $\mathsf{t} = 360\,\mathrm{s}$ the PandaBoard provides 25 949 bit per 32 kB of DRAM, versus 9692 bit for the Intel Galileo. At larger t more DRAM cells get a chance to decay, increasing l_{t} and hence the entropy. However, large values of t make PUF handling too



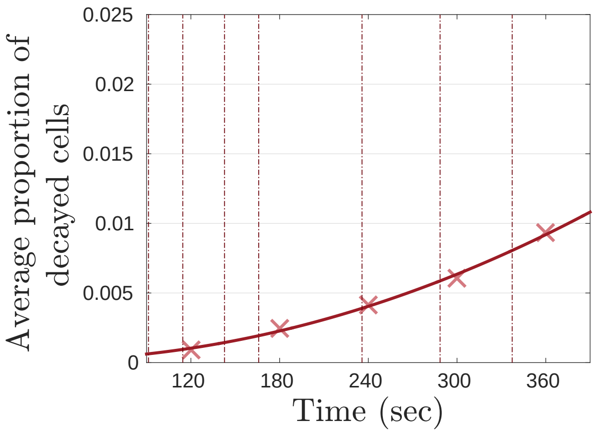
PandaBoard, measured at $\mathcal{T}_1$ and `size` = 16 MB.



Intel Galileo, measured at $\mathcal{T}_1$ and `size` = 16 MB.



PandaBoard, measured at $\mathcal{T}_2$ and `size` = 32 kB.



Intel Galileo, measured at $\mathcal{T}_2$ and `size` = 32 kB.

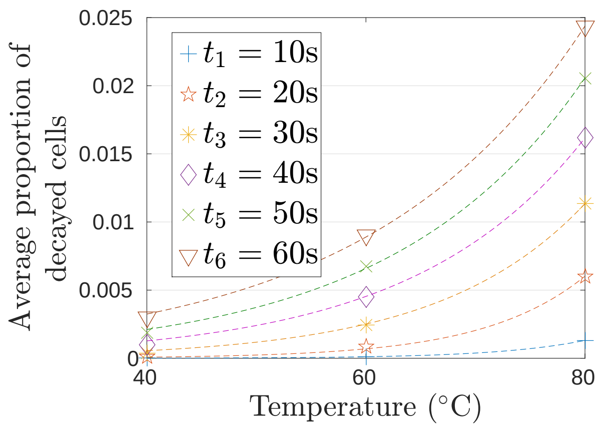
Figure 4.6.: Time-dependency of the decay DRAM cells on the PandaBoard and the Intel Galileo for $\mathcal{T}_1$ and $\mathcal{T}_2$, as well as `size` = 16 MB and 32 kB. Possible challenges are indicated by vertical lines.

slow to be practical. Regarding the fractional entropy, the values of the proposed DRAM PUF are orders of magnitude smaller, compared to SRAM PUFs, which usually have 0.7 bit to 0.9 bit of entropy per cell. However, DRAM usually provides significantly more memory cells than SRAM, and thus exhibits enough entropy in total.

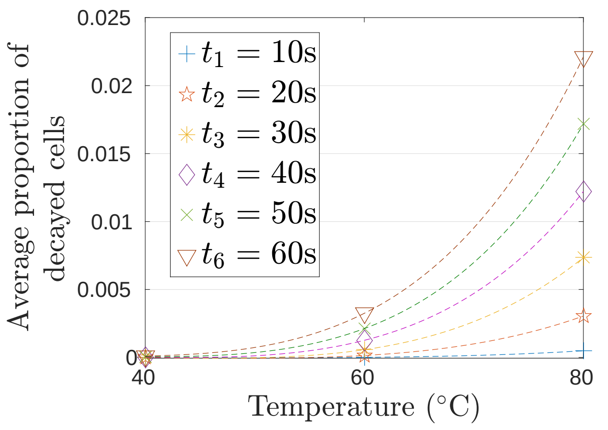
Decay dependency on time and temperature

Figure 4.6 shows the average proportion of decayed cells, l_{t}/N, as a function of time t. All measurements were taken at (ambient) room temperature with DRAM chips operating at around 40 °C. Every point in the plot represents an average taken over all logical PUFs. We see that the number of decayed cells significantly increases over time.

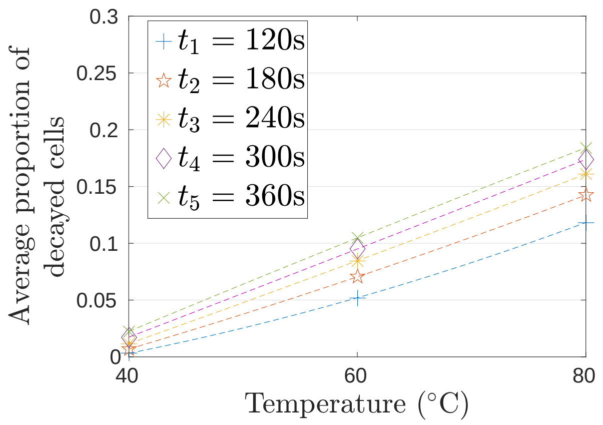
This plot allows us to estimate the number of time-dependent challenges that a logical PUF can support. In order to allow for unique identification at different decay times, the set of decay times $\mathcal{T} = \{\mathrm{t}_1, \mathrm{t}_2, ..., \mathrm{t}_n\}$ must be chosen so that the corresponding measurements taken



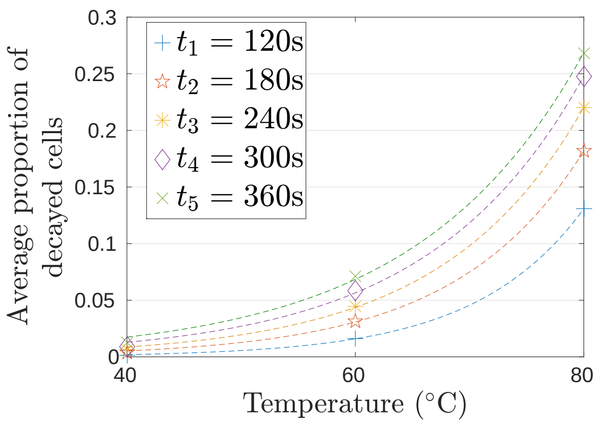
PandaBoard results, obtained at $\mathcal{T}_1$, using `size` = 16 MB.



Intel Galileo results, obtained at $\mathcal{T}_1$, using `size` = 16 MB.



PandaBoard results, obtained at $\mathcal{T}_2$, using `size` = 32 kB.



Intel Galileo results, obtained at $\mathcal{T}_2$, using `size` = 32 kB.

Figure 4.7.: Proportion of decayed DRAM cells as a function of temperature for the PandaBoard and the Intel Galileo for different decay times $\mathcal{T}_1$ and $\mathcal{T}_2$ and values of PUF `size` parameter.

at decay time t_{x+1} show a minimum number of *new* bit flips $\epsilon_{\mathrm{t}_x} = l_{\mathrm{t}_{x+1}} - l_{\mathrm{t}_x}$, with respect to the previous one t_x, which must be greater than the inherent noise. Given the noise values and ϵ_{t}, the set of viable decay times and thus the challenges of a logical PUF can be determined accordingly. We computed a conservative, minimum number of possible challenges per logical PUF, by using the maximum noise (i.e., minimum $\mathbf{J}_{intra}$ value) and the minimum number of bit flips, previously observed at each decay time t. We experimentally determined the maximum number of challenges for decay times $\mathcal{T}_1$ and a PUF size of 16 MB to be $n = 5$ for the Intel Galileo and $n = 6$ for the PandaBoard, as well as $n = 7$ and $n = 2$, respectively for $\mathcal{T}_2$ and a PUF size of 32 kB. Possible challenges are indicated by vertical red lines in Figure 4.6

A second factor influencing the number of decayed DRAM cells is temperature. Figure 4.7 shows the proportion of decayed cells as a function of temperature. The temperature was controlled using a heater circuit. We observed that heating DRAM affects all cells in the same way: the decay is accelerated by the same factor. At temperature $T' > T$ it is possible to find decay time $\mathrm{t}' < \mathrm{t}$ so that $s(\mathrm{t}')_{T'} = s(\mathrm{t})_T$, i.e., from a heated DRAM, operating at temperature T', the same response can be obtained in time t' as from a cooler DRAM in time t. Figure 4.8

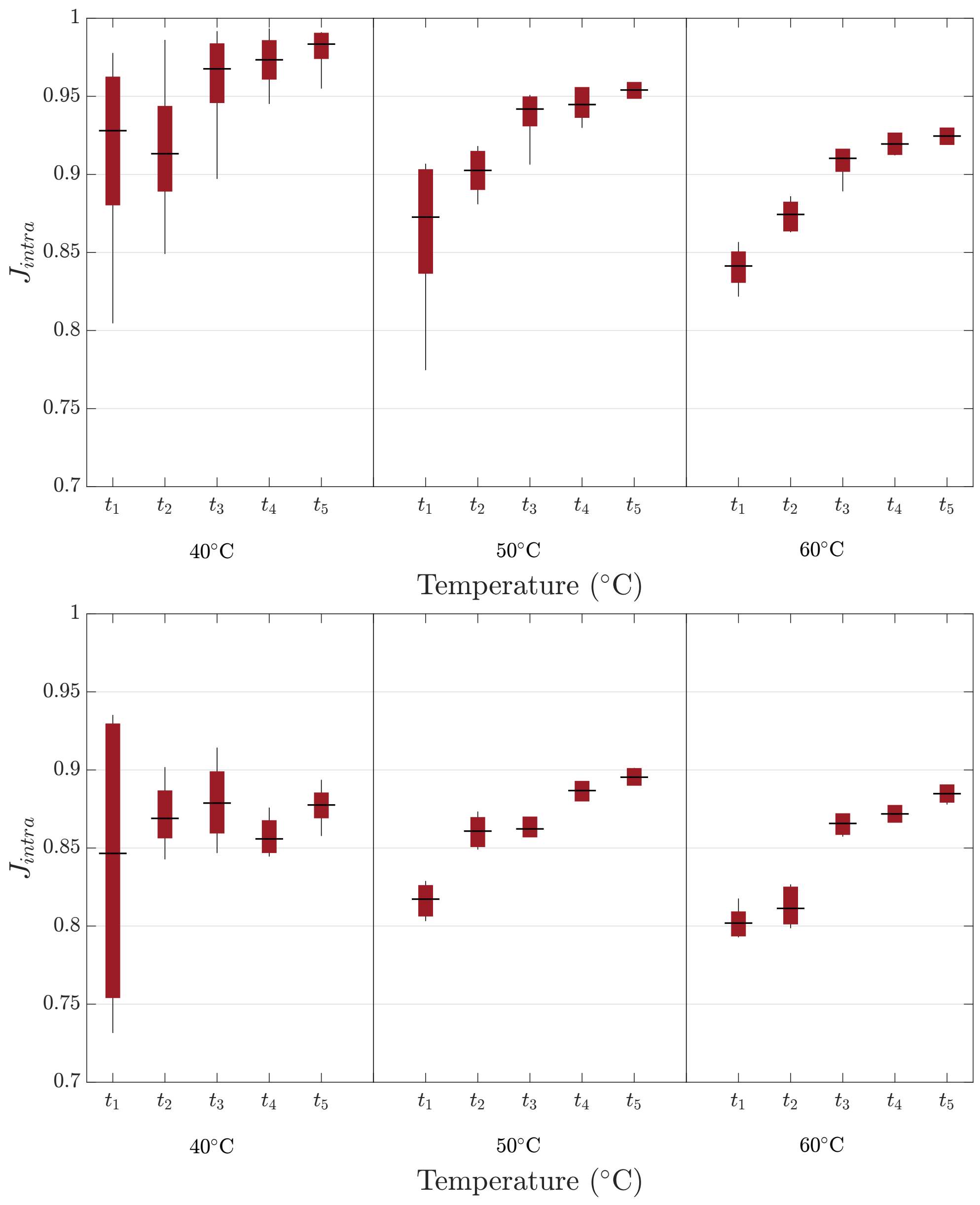
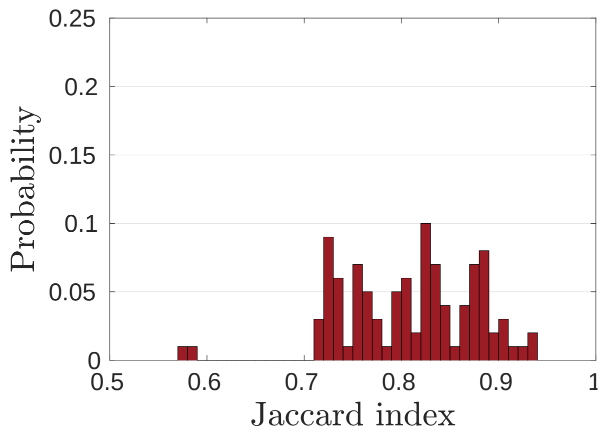


Figure 4.8.: $\mathbf{J}_{intra}$ values (i.e., similarity) of enrollment measurements taken at room temperature and measurements at higher temperatures $T' = \{40\,°\text{C}, 50\,°\text{C}, 60\,°\text{C}\}$, with adjusted decay times t' for the PandaBoard (top) and Intel Galileo (bottom).

shows the Jaccard index $\mathbf{J}_{intra}$ between measurements $s(\mathsf{t})_T$ at room temperature $T = 40\,°\text{C}$ and measurements $s(\mathsf{t}')_{T'}$ performed on the same PUF, with adjusted t' for different temperatures T'. We used $t = \{120\,\text{s}, 180\,\text{s}, 240\,\text{s}, 300\,\text{s}, 360\,\text{s}\}$ and $T' = \{40\,°\text{C}, 50\,°\text{C}, 60\,°\text{C}\}$. For all measurements obtained from both device types, $\mathbf{J}_{intra}$ lies within the usual noise level. This confirms that differences in temperature can effectively be accommodated by adjusting t accordingly and that the PUF behavior at different temperatures can be predicted. The adapted decay time t' at an increased temperature T' that results in a similar decay behavior as using decay time t at temperature T (with $T' > T$), can be modeled as:

$$\mathsf{t}'_{T'} = \mathsf{t} \cdot e^{-\alpha(T'-T)}. \tag{4.6}$$



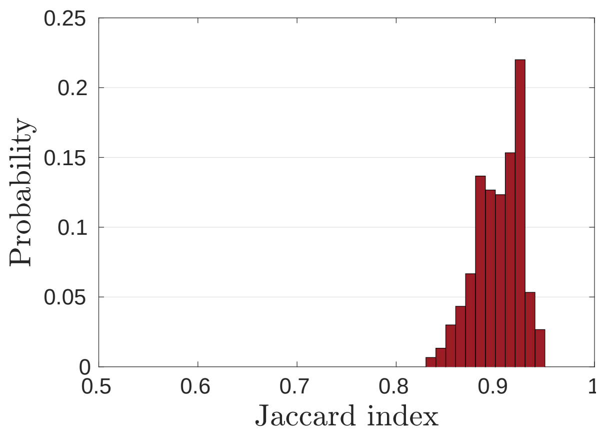


Figure 4.9.: Distribution of $\mathbf{J}_{intra}$ values computed between measurements pairs, taken at enrollment and reconstruction from the same logical PUF instances, roughly 16 months apart. Values are shown for the PandaBoard (left) and Intel Galileo (right).

Based on our measurements, we estimated α to be 0.066 for the Intel Galileo platform and 0.068 for the PandaBoard. The estimation of both models is given for a confidence interval of 0.95.

Stability over time

During the extended lifetime of devices, DRAM aging effects will begin to take place. Existing work on SRAM PUFs has explored aging effects. [ML14; Mae+12; SŠK15; Sel+11] We are only aware of limited work on aging-related effects in DRAM cells with regard to security [SPW09]. Figure 4.9 shows the histogram of $\mathbf{J}_{intra}$ values for measurements of both evaluation boards, taken roughly 16 month apart. Note that the measurements also include the noise introduced by temperature changes in our lab. The values for the Intel Galileo and the Galileo are similar to the $\mathbf{J}_{intra}$ results shown in Table 4.7, suggesting sufficient stability of DRAM PUFs over a long-term usage time period.

4.5.3. Rowhammer PUF Results

The evaluation of the Rowhammer PUF follows a two-staged process. First, the main goal is to maximize the number of bit flips, i.e., the entropy of the puf measurements, based on different parameter settings. Therefore, we will first discuss how different values of the parameters presented in Section 3.3.2 affect the number of observed bit flips. Secondly, we evaluate the Rowhammer PUF on the basis of the parameter configuration that maximizes the number of bit flips, regarding uniqueness, robustness and entropy. Lastly, we discuss varying ambient temperature conditions that could influence the Rowhammer PUF.

We will follow an explorative approach, which involves assessment of a subset of all potential parameter values. Due to the lack of information about the distribution of true- and anti-cells[4], it is necessary to explore the correlation between parameter values and PUF behavior experimentally, by testing various parameter settings. One approach to retrieve the layout of true cells (and anti-cells) is to initialize the DRAM with ‘`0xFF`’ (‘`0x00`’), disable DRAM refresh and read-back the memory contents after a period of several hours or days, i.e. the end of the decay process.

[4] Usually this is the case when dealing with COTS devices as most vendors treat such fine-grained implementation details regarding their hardware components as intellectual property and thus will not disclose details.

Parameter	Evaluated Values
`RH type`	single-sided (`SSRH`), double-sided (`DSRH`)
`RH size`	4 kB, 32 kB, 128 kB
`Hammer row iv`	'`0x00`', '`0x55`', '`0xAA`', ' `0xFF`'
`PUF row iv`	'`0x00`', '`0x55`', '`0xAA`', ' `0xFF`'
`RH time`	60 s, 120 s

Table 4.5.: Parameters used during evaluation of the Rowhammer PUF characteristics and their corresponding set of values.

In our evaluation, three different memory regions, each located on one individual PandaBoard, are measured.[5] For all the measurements, the PUF address was fixed. For each parameter combination, 20 measurements were taken.

The bit flips we observe in the Rowhammer PUF measurements are due to

I. the hammering process, as well as the

II. DRAM cell decay that emerges after DRAM refresh is disabled (cf. Section 4.3.3).

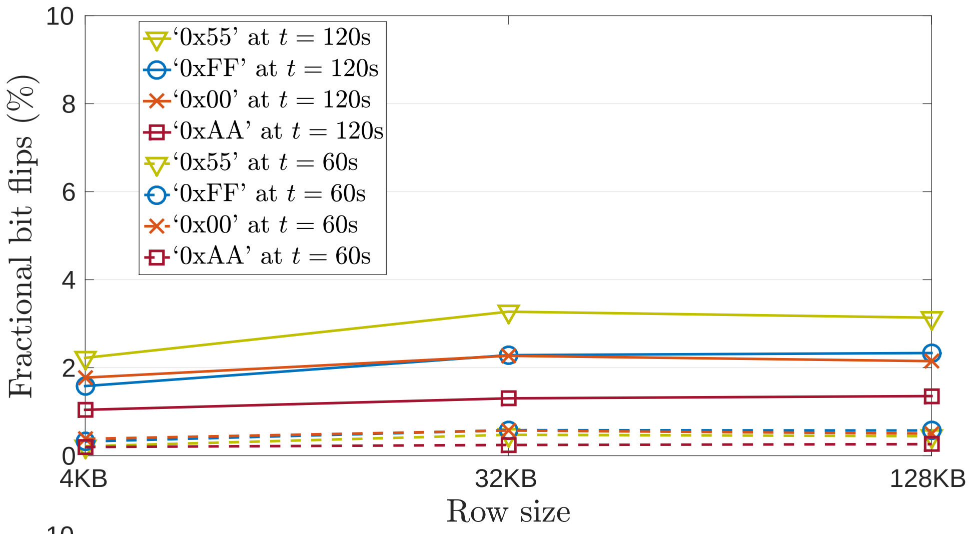
In order to confirm that Rowhammer adds a significant number of extra flips, we measured the number of bit flips, which are solely caused by the decay process, and compared it to the total number of bit flips which we observed in the Rowhammer PUF measurements. Compared to the bit flips caused by DRAM decay, the Rowhammer PUF introduces 2.4 times bit flips in 60 s and about twice the number of bit flips in 120 s. Further, the set of bits that flip (i.e., their locations in the measurements) obtained from the Rowhammer PUF partially overlap with the set of bit flips induced by the DRAM decay process, even for longer decay times (i.e., without the influence of the Rowhammer effect). Thus, the Rowhammer PUF induces new bit flips, which are at different locations compared to the DRAM decay process.

In order to extract the maximum possible entropy from PUF measurements, we primarily strive to maximize the number of bit flips. For this purpose, we first identify those parameters that have the largest effects on the amount of bit flips. In the following paragraphs, we will discuss the parameters listed in Table 4.5, in the context of their impact on observable bit flips.

`RH type`: Initially, we expected the `RH type` parameter to have a strong influence on the number of flipped bits. In Figure 4.10, we present the *fractional* number of bit flips as a percentage of the absolute number of bits available (`RH size`). Contrary to our expectations, applying `DSRH` (right) instead of `SSRH` (left), does not lead to a highly increased number of flips, despite hammering both rows adjacent to respective PUF rows. Instead, compared to `SSRH`, using `DSRH` only leads to approximately 9 % more bit flips in 60 s and about 15 % in 120 s on average.

`RH size`: The `RH size` influences the total time required to execute a single iteration of hammering the DRAM. In our implementation, each hammer row is accessed roughly every 6 μs when hammering 2 rows (4 kB PUF) and every 8 μs when hammering 17 rows (128 kB PUF) in the `SSRH` setting. Further, Figure 4.10 shows the number of bit flips relative to `RH size`. The number of bit flips does not change significantly for different values of `RH size`, i.e., the fraction of bit flips for different memory ranges stays stable.

[5] Hereafter, we denote such a memory region as a *PUF instance*.



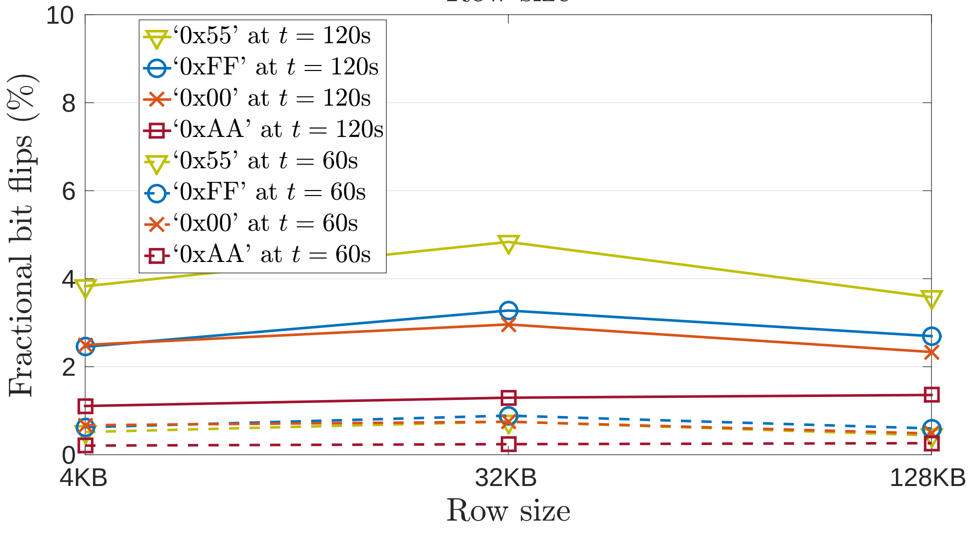


Figure 4.10.: Fractional number of bit flips, given in percent relative to `RH size`, using `PUF row iv` =‘`0xAA`’. Number of bit flips using `SSRH` (top) and using `DSRH` (bottom).

`Hammer row iv` and `PUF row iv`: Given that DRAM arrays consist of true-cells and anti-cells, the initial value (IV) of the hammer rows as well as the PUF rows is expected to play an important role regarding the number of observable bit flips. Depending on the type of cell, a bit flip in a PUF row can be observed only if the cell is initialized with the logic value that corresponds to its charge state. Similarly, due to the physical interaction of charged analog elements in the hammer and PUF rows (i.e., wires and capacitors) and the resulting charge interaction paths, different IVs of the hammer rows influence the number of bit flips as well. Thus, the values of both parameters must be chosen carefully, in order to maximize bit flips. As can be seen in Table 4.6, different configurations of `Hammer row iv` and `PUF row iv` lead to measurements that exhibit different bit flips. In general, it can be inferred from the experiments that the number of bit flips on the PandaBoard can be maximized, if PUF rows are pre-initialized in such a way that keeps true-cells and anti-cells in the charged state, whereas cells of the adjacent hammer rows are kept in an uncharged state. In particular, the measurements show that most bit flips can be observed, if PUF rows are initialized with ‘`0xAA`’, which depicts a bit-wise checkerboard pattern with a leading ‘1’. Instead, adjacent hammer rows are set up using the complementary pattern, starting with a ‘0’ bit (‘`0x55`’). In contrast, no bit flips can be observed when initializing

PUF row iv	Hammer row iv ‘0x00’	‘0x55’	‘0xAA’	‘ 0xFF’
’0x00’	7405 / 8032	17558 / 20358	7391 / 7200	17288 / 20152
’0x55’	0 / 0	0 / 0	0/0	0 / 0
’0xAA’	22547 / 24480	**32904** / **37548**	14218 / 14243	24479 / 28268
’0xFF’	15633 / 17798	15402/15402	6132 / 6479	6095 / 6416

Table 4.6.: Overview of the average number of observable bit flips, depending on combinations of `Hammer row iv` and `PUF row iv`. Configuration used: `RH size` = 128KB and `RH time` = 120s (`SSRH`/`DSRH`).

PUF rows with ’`0x55`’. We hence reason that in this case cells of the PUF rows are initialized in correspondence with their uncharged states.

Optimal Parameter Configuration: The parameters `Hammer row iv` and `PUF row iv` have a predominant influence on the number of bit flips, which we strive to maximize. We therefore first fix their values as follows: `Hammer row iv` =‘`0x55`’ and `PUF row iv` =‘`0xAA`’. We further set `RH type` to `SSRH`, as the number of introduced bit flips is in the same order of magnitude as `DSRH`. Instead, `SSRH` requires ≈ 55% less memory and involves less memory accesses compared to `DSRH`. Further, we will use `RH time` = 120 s, as we achieve ≈ 400% more bit flips as for 60 s. Although the resulting PUF readout time is similar to existing runtime accessible DRAM PUFs, we will look at improving the time in future work.

PUF Characteristics

In order to assess the applicability of the set of flipped bits as a PUF, we validated uniqueness, robustness and entropy of the Rowhammer PUF measurements, using the parameter configuration identified above. Similar to the evaluation of the decay-based DRAM PUF, we utilize metrics based on the Jaccard index (cf. Section 4.4.2), instead of the Hamming distance. This again is motivated by the fact that DRAM-based PUFs show different characteristics compared to classic PUFs (such as SRAM-based PUFs). In particular, measurements from DRAM modules draw their PUF characteristics from the location of the flipped bits. This fact (uniqueness of indices) is not properly reflected in Hamming distance-based measures.

Decay Times $\mathcal{T}$	`RH type`	min. $\mathbf{J}_{intra}$	max. $\mathbf{J}_{inter}$	H_t (bits)	avg. Number Bit Flips
$\mathcal{T} = \{60\,\text{s}, 120\,\text{s}\}$, `Hammer row iv` =‘`0x55`’, `PUF row iv` =‘`0xAA`’, `RH size` = 4 kB					
60 s	SSRH	0.796	0.003	0.045	162
	DSRH	0.871	0.011	0.087	359
120 s	SSRH	0.900	0.009	0.154	730
	DSRH	0.891	0.030	0.234	1256
$\mathcal{T} = \{60\,\text{s}, 120\,\text{s}\}$, `Hammer row iv` =‘`0x55`’, `PUF row iv` =‘`0xAA`’, `RH size` = 32 kB					
60 s	SSRH	0.939	0.008	0.075	2405
	DSRH	0.943	0.010	0.104	3581
120 s	SSRH	0.972	0.025	0.208	8583
	DSRH	0.975	0.032	0.279	12679
$\mathcal{T} = \{60\,\text{s}, 120\,\text{s}\}$, `Hammer row iv` =‘`0x55`’, `PUF row iv` =‘`0xAA`’, `RH size` = 128 kB					
60 s	SSRH	0.945	0.008	0.068	8537
	DSRH	0.947	0.005	0.066	8259
120 s	SSRH	0.966	0.025	0.201	32904
	DSRH	0.965	0.020	0.223	37548

Table 4.7.: Evaluation results of the Rowhammer PUF using fixed initialization values `Hammer row iv` =‘ `0x55`’ and `PUF row iv` =‘ `0xAA`. Results are given for `RH time` $\mathcal{T} = \{60\,\text{s}, 120\,\text{s}\}$, `RH size` $= \{4\,\text{kB}, 32\,\text{kB}, 128\,\text{kB}\}$, and both variants of `RH type`.

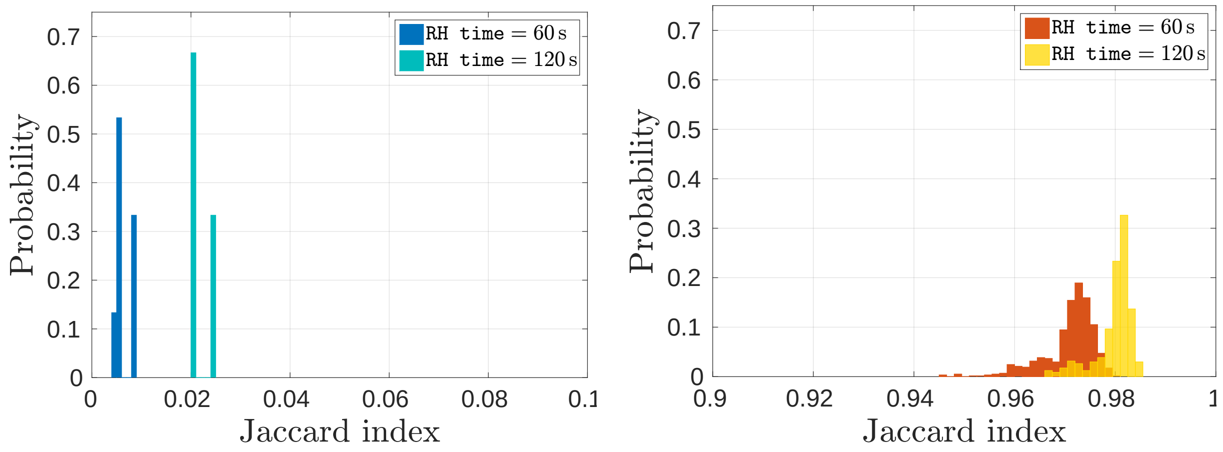


Figure 4.11.: Histograms of $\mathbf{J}_{inter}$ (left) and $\mathbf{J}_{intra}$ (right) values for three PUF instances using 20 measurements with `PUF row iv` =‘`0xAA`’, `Hammer row iv` =‘`0x55`’, `RH size` = 128 kB and `RH type` set to `SSRH`.

Uniqueness and Robustness

Figure 4.11 shows the histograms for values obtained for uniqueness and robustness metrics, using the optimal configuration set and `RH size` = 128 kB. Clearly, both histograms separate, indicating that Rowhammer PUF instances can be robustly and uniquely identified. With a minimum $\mathbf{J}_{intra}$ value of 0.9454 at 60 s, and a maximum noise of ≈ 3% at 120 s, Rowhammer PUF measurements show high uniqueness that allow for discriminating between different PUF instances, and a high robustness, so that standard Fuzzy Extractor constructions [DRS04] can be applied for reliable key reconstruction.

Entropy

PUF measurements should exhibit sufficient entropy in order to derive a cryptographic key. We estimated the Shannon entropy $\mathbf{H}(X)$ of the PUF measurements, similarly to the decay-based PUF. Denoting the total number of bits contained in a PUF measurement (i.e., `RH size`) as N, l as the cardinality of s_x and assuming that the bits flip independently from each other, the Shannon entropy can be calculated as follows:

$$\mathbf{H} = \log \binom{N}{l}. \tag{4.7}$$

Using the minimum number of bit flips ($l = 30994$) observed in the measurements, based on the optimized parameter setting identified above ($N = 1048576 \mathrel{\hat{=}} 128\,\text{kB}$), the lower bound for the *fractional* Shannon entropy (i.e., the entropy per cell), is 0.192. Given the vast amount of available cells, the PUF measurements show sufficient entropy to derive cryptographic keys. For example, the derivation of a 128 bit key, given a fractional entropy of 0.192, requires approximately 85 B, while the PUF is already 128 kB (excluding entropy required to compensate leakage due to Helper Data).

Temperature Dependency

The behavior of DRAM bit flips can be influenced by the operating temperature. In order to validate usage of the Rowhammer PUF in several operating temperatures, we evaluated the PUF at temperatures of 40 °C (working temperature of DRAM on PandaBoard), 50 °C and 60 °C. We computed the number of bit flips and $\mathbf{J}_{intra}$ values for measurements taken at these respective temperatures. Table 4.8 shows the average number of bit flips as well as the minimum $\mathbf{J}_{intra}$ (i.e., maximum noise). The temperature evaluation shows that, while bit flips increase at higher temperatures, the noise level stays constant for each temperature. Thus, the Rowhammer PUF exhibits sufficient stability to be used at higher temperatures.

Metric	Operational Temperature		
	40 °C	50 °C	60 °C
avg. bit flips	32904	65431	132450
min. $\mathbf{J}_{intra}$	0.966	0.981	0.985

Table 4.8.: Comparison of the average number of bit flips and minimum $\mathbf{J}_{intra}$ values at operating temperatures of 40 °C, 50 °C and 60 °C using the optimal parameter configuration.

4.6. Chapter Summary

SRAM-based PUF

The most important conclusion that can be drawn from the evaluation results is that PUF behavior can be found in the SRAM modules of many commercially available MCU platforms. Most of the SRAMs that have been measured show promising results and therefore are suitable for use in PUF implementations. However, the amount of pre-processing required on the data will vary between the platforms. In order to give advice on the usability of an SRAM module in a PUF scenario one must also take into account the available SRAM size. This is due to the fact that during the pre- processing steps usually a certain amount of entropy is lost due to biases of the start-up values, the level of noise as well as the so-called Helper Data stored publicly. Thus, although a device type might have good PUF characteristics with respect to the individual metrics, it might not be useful as a PUF instance since the SRAM size available is not enough to compensate for entropy loss exhibited during pre-processing. Furthermore, in many application scenarios it is desired to save as much as possible left-over SRAM, which is not used for PUF usage (i.e., key reconstruction). This is due to the fact that the actual code for reading the SRAM start-up values and reconstructing the key also requires space on the SRAM. Thus, the total available size of the SRAM has an influence on the applicability of the on-chip SRAM of a given device type. Finally, the results of the aging experiments show that aging indeed causes drifting of the PUF measurements over time. In particular, our study of experimental data confirms the existence of asymmetric (data-dependent) drift in SRAMs PUFs. This drift must be considered if the PUF is designed for long-term usage. Further, systematic drift can invoke issues in the context of privacy-preserving protocols. Therefore, an exhaustive aging analysis of a given SRAMs PUF instance is fundamental, especially in the context of privacy-related applications.

DRAM-based PUF

The evaluation of the DRAM PUFs found on unmodified, commodity devices, in particular the PandaBoard and Intel Galileo, showed their high robustness, uniqueness and randomness. Using the approach of adaptive decay times, it is further possible to use the decay-based DRAM PUFs in different ambient temperature conditions. Time stability results further suggests that a given platform implementing this PUF type can be robustly used over longer periods. Our intrinsic DRAM PUFs overcome two limitations of the popular intrinsic SRAM PUFs: they have the ability to be accessed at runtime, and have an expanded challenge-response space due to the use of a decay time t that is part of the challenge. Consequently, our work presents a new alternative for device authentication by leveraging DRAM in commodity devices.

Rowhammer PUF

Similarly to the decay-based DRAM-based PUF, the Rowhammer PUF also shows favorable PUF characteristics. Almost perfect robustness and uniqueness results allow for reliable extraction of device-unique cryptographic keys from the dram bit flip patterns. Furthermore, given good entropy results and the large amount of DRAM available on many present and future low-cost MCUs, allows for extraction of a high number of different keys. Temperature and time stability measurements suggest that this novel PUF type can be robustly used over larger periods of time and at higher operational temperatures. Unlike the majority of work that has used the Rowhammer effect to trigger a security exploit, this novel PUF type displays the first positive usage of the Rowhammer effect. Further, the Rowhammer PUF is one of the few intrinsic memory-based PUFs (next to the DRAM-based PUF) that can be challenged at runtime.

This section showed that standard memory components, i.e., SRAM and DRAM, which are implemented on COTS MCUs, exhibit manufacturing variations that can be used for device identification. On the basis of an empirical analysis of various wide-spread device types, it was shown that these variations can be successfully extracted and leveraged for PUFs usage, in order to provide a robust and reliable hardware-based trust-anchor. In the following chapter we build on these intrinsic PUF-based trust-anchors to design various lightweight security protocols and applications.

Chapter 5

Secure Boot

Published Content

- André Schaller, Tolga Arul, Vincent van der Leest, and Stefan Katzenbeisser. *Lightweight Anti-counterfeiting Solution for Low-End Commodity Hardware Using Inherent PUFs.* In: *Trust and Trustworthy Computing.* 2014, pp. 83–100.
- Anthony Van Herrewege, André Schaller, Stefan Katzenbeisser, and Ingrid Verbauwhede. *DEMO: Inherent PUFs and Secure PRNGs on Commercial Off-The-Shelf Microcontrollers.* In: *Proceedings of the 2013 ACM Sigsac Conference on Computer & Communications Security.* 2013, pp. 1333–1336.

Most of the commercially available embedded devices lack the implementation of dedicated security mechanisms such as a Trusted Execution Environment (TEE), as shown in Chapter 2.2.1. Despite the poor or often non-existing security features found in embedded devices, they are nevertheless used frequently to process sensitive data, i.e., as part of industrial automation [SWW15] or as part of safety-critical environments such as car-2-x scenarios [Ger+14; NJ16]. Due to the combination of low security levels of such devices at best and the fact that many are used to store and compute information that are valuable assets to many cyber-criminals, they have become a worthwhile target of adversaries.

A malicious party has multiple incentives to attack deployed embedded devices. A highly profitable motive is to extract intellectual property (IP) stored on a device in the form of the firmware application and related secret assets such as cryptographic keys or sensitive data. After successful extraction the attacker is able to deploy the IP on counterfeit devices. Alternatively, he could deploy an alternative firmware application which was not intended for execution on the given device. In particular, the attacker might want to circumvent the vendor's licensing model by manipulating the firmware application. To evade licensing restrictions the attacker could try to downgrade to a previous firmware version, to exploit design flaws and consequently escalate privileges on the system. Another motivation to alter the firmware application is to capture valuable user data like passwords, credentials and usage data. Furthermore, the attacker might be able to actively manipulate output data, as for example in the case of smart metering devices, in order to report fake consumption data. While the integrated design of today's Microcontroller Units (MCUs) can protect firmware, stored in on-chip Non-Volatile Memory (NVM) from read-out attempts, some commercial platforms implement their NVM instances off-chip (i.e., the PandaBoard) and hence expose firmware to rogue extraction. Moreover, hardware developers often release embedded devices without disabling debugging ports, such as Joint Test Action

Group (JTAG). Usually this is done on purpose to trade security in for firmware reconfigurability (i.e., to upgrade firmware versions). However, access to debugging ports effectively allows adversaries to extract or replace firmware images, even from on-chip NVM [Goo09; RK10].

To overcome the issue of malign extraction of the firmware application, we propose a lightweight solution to securely boot an encrypted firmware application from within a trusted immutable boot loader. By requiring the decryption of the application using a *device-dependent* key prior to its execution, our solution effectively protects from attempts to extract the firmware application from NVM and realizes strict hardware-software binding between the application and the underlying embedded platform at the same time. For this purpose the scheme relies on a hardware-based anchor of trust in terms of Physically Unclonable Function (PUF) instance. It uses intrinsic memory-based PUFs extracted from the on-chip Static Random-Access Memory (SRAM) module, in order to derive a hardware fingerprint, which is unique for individual devices. The fingerprint is further processed during an early boot stage to generate an ephemeral cryptographic key and to subsequently decrypt the firmware. Our scheme establishes trust in the on-chip hardware and in the firmware application executed on the device by linking both instances. We achieve protection against firmware IP extraction or its replacement on embedded devices without dedicated security mechanisms. As we will show in Section 5.4, the solution is compatible with off-the-shelf commodity MCUs and System on Chips (SoCs).

In the following sections, we describe the considered threat model, present the proposed security architecture and explain the course of usage. Lastly, we describe details related to the implementation of the proof-of-concept that prove the applicability of our scheme.

5.1. Threat Model

We consider an economically-driven, simple hardware adversary $\mathcal{A}$, who is able to inspect or replace any peripherals, i.e., such hardware components, that are off-chip and hence outside the MCU security boundary (cf. Figure 2.1). We disregard any invasive or other advanced physical attacks as such attacks generally require expensive equipment or laborious engineering efforts to reverse engineer the hardware layout [Sko12], which quickly renders such attempts uneconomic in the case of low-cost embedded devices. The attacker hence can read out the contents of the external memory (usually Dynamic Random-Access Memory (DRAM) or Electrically Erasable Programmable Read-only Memory (EEPROM)) as it is highly exposed to external accesses. Thus, we imply that $\mathcal{A}$ can read out and subsequently change the firmware that is stored on external memory. We further consider $\mathcal{A}$ to have access to active interfaces to the MCU itself, i.e., by means of enabled debugging ports. $\mathcal{A}$ is thus able to extract or replace the firmware image, even in case it is stored on on-chip NVM. In addition, $\mathcal{A}$ is able to inspect and modify on-chip memory values with software of his choice after the boot process. For the attacker to achieve one of the mentioned goals we consider $\mathcal{A}$ to have the following abilities. The attacker has physical access to the device due to its ubiquitous availability or because the attacker possesses the device as a legitimate user.

Besides these capabilities, we assume that the attacker cannot perform one of the following actions. The attacker is not able to change the code of the boot loader as it is stored in a Read-Only Memory (ROM), which is under control of the manufacturer. Furthermore, we consider an attacker not to be able to replace the ROM chip with a second one of his choice, containing boot code under his control. Especially in the case of SoC platforms on-chip memory is highly integrated and a replacement of a memory module is beyond the means of the average-skilled attacker. Lastly, the attacker is not able to read out the start-up values of the on-chip SRAM during start-up. The start-up values are protected by the boot loader and are erased shortly after the device gets out of reset. As soon as the device is powered the boot loader reads the start-up values and immediately overwrites them before the firmware is called. The boot loader is assumed to be trusted and cannot be replaced. Hence, the first possibility for the attacker to execute code of his choice is after the boot loader finished execution.

We are aware of the fact that a physical attacker in possession of sufficient resources in terms of time and money can circumvent virtually any security mechanism. Nevertheless, if the attacker would succeed to extract the SRAM start-up values, i.e. the cryptographic key, he would only be able to attack this individual device and has to perform the same attack for any other device.

5.2. Architecture

The proposed anti-counterfeiting solution is designed for implementation on a variety of commodity hardware without on-board security facilities ranging from low-cost devices to more complex SoC platforms (cf. Chapter 2.2.1). To be compatible with commercially available embedded devices and hence to be in line with the research goals of this thesis, our solution merely employs hardware components which are already present in virtually any computing device. In particular, we require the devices to be equipped with a programmable ROM to guarantee immutability of the customized boot loader, the processor containing the MCU itself, on-board SRAM (which is the source of the PUF instance) and external memory to store the encrypted firmware and so-called Helper Data as explained in Chapter 5.3. Furthermore, we assume that the manufacturer can modify the boot loader in order to modify the standard boot loader to implement the PUF interface, i.e., key extraction and decryption functionality of the firmware.

Differences in the boot loader architecture of class-0 as well as class-1 device types and more complex SoCs-based class-2 platforms (cf. Chapter 2.2.2) require slight modifications of the architecture of our proposed solution.

Generally, in order to preserve integrity of the boot loader, it needs to be immutable and thus must be stored ROM. Integrity of the boot loader must be provided as it queries the PUF by reading the SRAM start-up values and implements functionality to derive the cryptographic key. The key derived from the PUF measurement is used to decrypt the firmware application, stored in NVM, usually flash memory. In the case of more complex SoCs that leverage multi-staged boot loaders, the proposed architecture is slightly modified in the following way. Similar to the MCU case, the 1^{st}-stage boot loader is customized to implement the PUF interface, key extraction and decryption routines and hence must be immutably stored in ROM, in order to preserve its integrity. However, in the SoC case, the derived key will be used to decrypt the 2^{nd}-stage boot loader, instead of the firmware in the MCU scenario. Once the 2^{nd}-stage boot loader is decrypted and executed, it derives a second key, which is subsequently employed to decrypt the kernel image file. Eventually, the decrypted kernel can be executed, starting the operating system. Figure 5.1 illustrates the two cases.

The following paragraph explains the architecture for the SoC case in more detail as we used an SoC for our prototype implementation. In this scenario, the integrity of the 1^{st}-stage boot loader is guaranteed by programming it to a ROM module. It is executed as the first code after

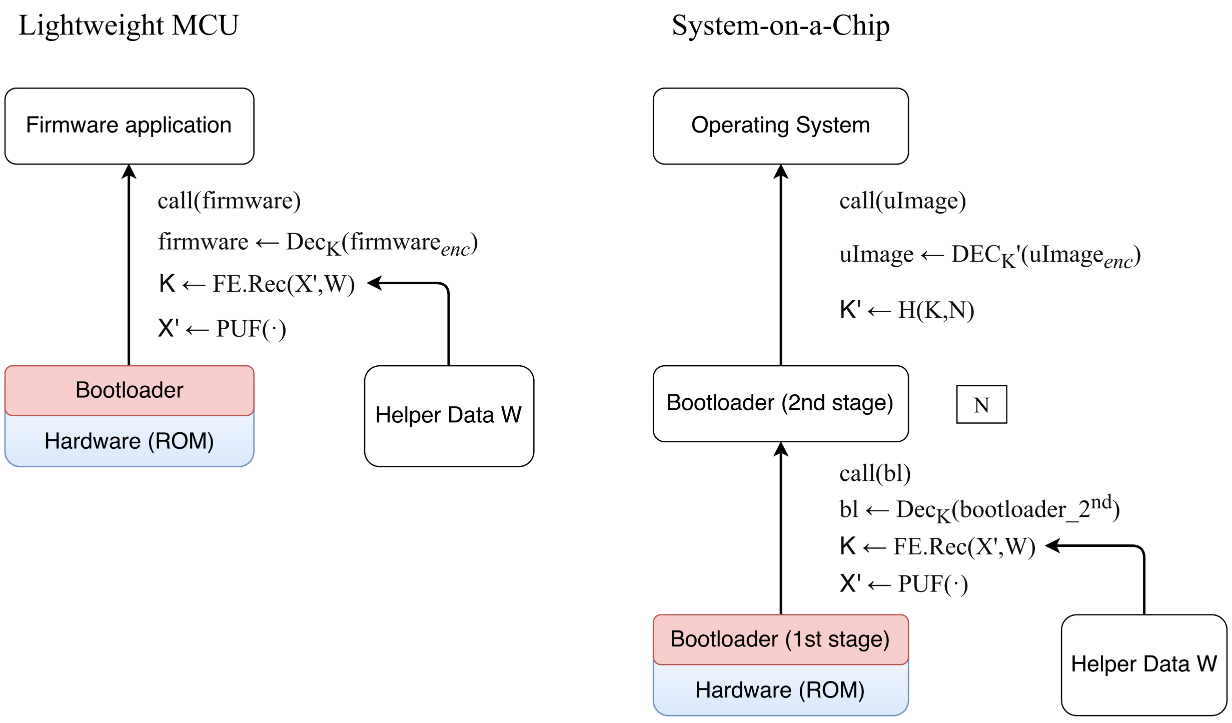


Figure 5.1.: Secure boot architecture for low-end MCU- and more complex SoC-based device types.

the device start-up[1]. It queries the SRAM PUF, deriving the device-dependent key K. The key exists in on-chip memory only for the period of the following two steps. K is used to decrypt the 2^{nd}-stage boot loader, stored in non-volatile memory (e.g., flash memory or DRAM).

After successful decryption of the 2^{nd}-stage boot loader, a second key K' is derived by hashing the concatenation of K and a salt value N: K' = **H**(K,N). Key K' is subsequently used to decrypt the compressed kernel file that also resides in NVM. The second key K' is derived to impede the reconstruction of K in case an attacker captures K'. If the attacker is able to capture K', a new version of the 2^{nd}-stage boot loader, including a new salt N_{new}, must be deployed which will generate a different N'_{new}. Furthermore, a new version of the firmware, encrypted under a new key K'_{new} must be distributed to regain a secure state. This design assures that only a 2^{nd}-stage boot loader can be executed that was encrypted by the correct device-depended cryptographic key K. Since the second key K' is derived from K also only such firmware can be properly loaded and executed, which was encrypted by the correct key as well. Thus, the operating system only boots properly, if the correct combination of hardware and software is in place.

5.3. Usage Process

The usage of the proposed secure boot architecture involves two phases, which align with the execution sequence of the underlying Fuzzy Extractor construction (cf.Section 3.1.3: i) the en-

[1] More precisely, on our implementation board the first code executed is a vendor-specific initialization code, which cannot be disabled and leads to a pre-initialized part of the level-3 on-chip SRAM on the PandaBoard

rollment phase of the SRAM PUF instance, performed by a trusted party, i.e., the manufacturer; and ii) the reconstruction phase which is conducted every time the user boots the embedded device. During both phases a Fuzzy Extractor algorithm operates on the SRAM start-up values, i.e., the PUF measurement, by executing `FE.Gen()` and `FE.Rec()` accordingly, in order to extract a reliable key.

5.3.1. Enrollment

The enrollment process is carried out by the manufacturer or system integrator and is performed once for each device. It serves two main purposes: the derivation of device-specific key K and the generation of so-called Helper Data W. The cryptographic key K is derived from a randomly chosen secret S, which is fed as input to a cryptographically secure hash function $\mathbf{H}(\cdot)$: K$\leftarrow$ $\mathbf{H}$(S). The secret S is predefined by the manufacturer and must be unique for every device.

The Helper Data W will be used later during the reconstruction phase to reconstruct the secret S and subsequently to derive the key K, given a noisy SRAM measurement X'. W is stored in external memory as it does not leak information about S. Furthermore, to protect the Helper Data from tampering several methods can be applied, such as the approach described by Boyen [Boy04].

5.3.2. Reconstruction

The reconstruction process is performed at the user's side and is executed every time the user boots the corresponding device. After powering the device, the 1^{st}-stage boot loader reads and stores the noisy SRAM start-up values X' and immediately overwrites the start-up values to make them inaccessible to a physical attacker. In a second step the boot loader reads the Helper Data W from external memory. The Fuzzy Extractor XORs X' with W and decodes the output using the concatenated decoders to construct the secret S'. The generated secret S' will only be equal to the correct secret S if the Helper Data corresponds to the respective device having SRAM start-up values X' that are similar to the enrollment measurement: $\mathbf{HD}$(X,X') $< \epsilon$. Next, the key K is derived by hashing secret S', which in turn is used to decrypt the firmware or the 2^{nd}-stage boot loader, depending on the scenario.

5.4. Proof of Concept

We implemented the proposed anti-counterfeiting architecture on the SoC-based PandaBoard. We chose an SoC platform for the following reasons. Primarily, we wanted to prove the feasibility to robustly extract a unique fingerprint also from more complex platforms. Furthermore, the boot process is more complicated compared to lightweight devices. Our intention was to show that the proposed solution can be implemented into existing boot loaders. Lastly, the size of the memory available on the SoC for implementing the Fuzzy Extractor logic is comparable to MCU-based device types as shown in Table 4.1. Hence, the requirements regarding memory footprint are equal to those of MCU-based devices. Thus, a successful implementation on SoCs proves the feasibility to implement the architecture on MCUs as well.

We used u-boot [Eng02], one of the most widely deployed boot loaders. It integrates a first-stage boot loader (also called Memory Locator — `MLO`) and a larger second-stage boot loader (`u-boot.img`). In the unmodified version of u-boot, the `MLO` performs minor hardware initialization as well as the setup of external DRAM. Afterwards it calls `u-boot.img` that is copied to DRAM memory. It initializes further hardware components and eventually calls the operating system kernel (`uImage`).

5.4.1. Enrollment

According to Section 5.3.1, the Helper Data W is derived from a randomly chosen secret S and a reference measurement X using a Fuzzy Extractor during the enrollment phase. The Fuzzy Extractor design is based on the construction presented by Bösch et. al [Bös+08] and will be explained in more detail in Section 5.4.3. We decided to set the size of the secret S to be 32 B. During the enrollment the key K is also generated by hashing the secret S with the SHA-3 hash function to a 256 bit key so that it can be used in the next step as input for the Advanced Encryption Standard (AES) implementation, configured to work with 256 bit keys. Following this, K is used to encrypt the `u-boot.img` using the AES-256 in CBC mode. The second key K' is used to encrypt the kernel image file (`uImage`). Eventually, the Helper Data as well as the encrypted files are stored in non-volatile memory.

5.4.2. Reconstruction

The main part of the reconstruction logic is implemented in the `MLO`, being one of the first pieces of code to be executed. As described in Section 5.3.2 the `MLO` reverses the flow of the enrollment phase by first extracting the on-chip memory chip's fingerprint X' and by processing it using the Fuzzy Extractor construction to derive the device-dependent key K. Subsequently, K is used to decrypt `u-boot.img`. Within the second-stage boot loader S' is computed, which is further used to decrypt the compressed Linux kernel file `uImage` that is executed eventually. More precisely, 256 B chunks of the decrypted `u-boot.img` are read into on-chip memory sequentially, get decrypted and are written back to external memory. After the successful decryption of `u-boot.img` is completed, it is executed. In case a false key K was generated a fault handler routine is called, displaying a warning message and canceling the boot process.

5.4.3. Fuzzy Extractor Design

To reproduce the secret key S from various noisy measurements error-correction is required. Following the suggestions of [Bös+08] we decided to implement a concatenated code comprising two linear codes – a Golay code and a repetition code – to reconstruct the 32 B secret. In particular, we are using a binary Golay-$(23, 12, 7)$ code in combination with a repetition code with 15 repetitions. Based on the model of a binary symmetric channel (BAC), the probability of observing a bit error is denoted as p_b. In contrast, the probability of receiving the originally sent bit without errors is $1-p_b$. According to [Gua+07], the probability that a bit string of length n, applied to the BAC model, observes more then t errors can be computed as:

$$P_{total} = \sum_{i=t+1}^{n} p_b^i (1-p_b)^{n-i} \binom{n}{i} = 1 - \sum_{i=t0}^{t} p_b^i (1-p_b)^{n-i} \binom{n}{i}. \tag{5.1}$$

Let s be the number of repetitions, ϵ be the average $\mathbf{HD_{intra}}$ and g be the number of Golay code words needed for decoding ($g = |\mathrm{S}|/12$). Hence, the error probability of the repetition code P_{rep} is calculated as:

$$P_{\mathrm{rep}} = \sum_{i=\lceil s/2 \rceil}^{s} \epsilon^i (1-\epsilon)^{s-i} \binom{s}{i}. \tag{5.2}$$

Similarly, the error probability of the Golay code can be computed as:

$$P_{\mathrm{Golay}} = \sum_{i=4}^{23} P_{\mathrm{rep}}^i (1-P_{\mathrm{rep}})^{23-i} \binom{23}{i}. \tag{5.3}$$

Accordingly and following the lines of [Bös+08], the bit error rate of the concatenated code, which resembles the False Rejection Rate (FRR) of the key construction process, can be calculated given the following equation:

$$P_{\mathrm{total}} 1 - (1 - P_{\mathrm{Golay}})^g. \tag{5.4}$$

With the construction at hand, we achieve a false rejection rate of 10^{-8} given an average $\mathbf{HD_{intra}}$ of 15 % as commonly used in literature [Bös+08]. The false rejection rate of the PandaBoard devices will be even lower since the measured $\mathbf{HD_{intra}}$ is well below the reference value of 15 % used in the calculations. Thus, the Fuzzy Extractor Design is suitable to reliably reconstruct a cryptographic key given several noisy SRAM PUF measurements for individual devices.

5.4.4. Evaluation

The overhead imposed by implementing the proposed scheme can be validated by considering the added code size and increased runtime during the boot stage. We measured the implementation used during reconstruction (cf. Section 5.4.2) as this functionality is employed at user side. Furthermore, the reconstruction implementation is slightly bigger compared to the enrollment logic. Table 5.1 lists the implementation overhead in terms of code size (Bytes) and runtime (milliseconds) of the respective functional blocks that compose the reconstruction implementation. In particular, during the reconstruction of our scheme, the following functions are sequentially processed: i) `readMeasurement`, which reads the SRAM start-up values as the PUF measurement and stores it in an array. The function `readHelperData` reads in the Helper Data. iii) the XORing of the PUF measurement and the Helper data is done in `measurementXOR`. iv) The repetition decoding, which is implemented as a majority voting scheme is handled by `decodeRepetition`, as well as v) `decodeGolay` that handles decoding of the Golay code. vi) The key S is created by means of SHA-3 hash function in `createKey`. Finally, vii) `u-boot.img` is decrypted and executed in `decryptUboot`.

Note that for evaluating runtime performance, we omitted overhead that is accounted to accesses to the SD card, which is the main NVM storage of the PandaBoard leading to disproportionately high I/O latencies that falsify runtime results. In fact, this approach of implementing flash memory is rather exotic, as most embedded platforms feature NVM either as part of the chip or as

Component	Runtime (ms)	Size (Bytes)
Base ROM	371	45477
`readMeasurement`	< 1	116
`readHelperData`	< 1	236
`measurementXOR`	1	160
`decodeRepetition`	3	342
`decodeGolay`	< 1	704
`createKey`	2	1140
`decryptUboot`	6944	1702
total	6951	4400

Table 5.1.: Implementation overhead with respect to runtime in milliseconds (left) and memory overhead in Bytes (right) for the PandaBoard.

peripheral memory, i.e., off-chip, for which accesses are orders of magnitude faster. In particular, we excluded timing results that are due to SD card accesses from `readHelperData` and `decryptUboot` functions.

5.4.5. Memory

In order to quantify memory consumption, we considered the static code segments (`.text`) and read-only data (`.rodata`) of our implementation. The results of Table 5.1 show that the main portion of the memory overhead accounts for the cryptographic functions, including the AES and Keccak implementations. The AES implementation, which is used during the decryption of the `u-boot.img`, requires 1702 kB on the ARM platform. To the best of our knowledge, we used one the currently most compact AES implementation, which is freely available [kok17]. The Keccak implementation requires roughly 1 kB and is based on the implementation of [Mar17]. A second contributor to memory overhead is the implementation for decoding the repetition and the Golay code words. Together they occupy approximately 1 kB. Besides cryptographic primitives, the decoding steps of the Fuzzy Extractor implementation, i.e., the Golay decoding claiming 704 B and the repetition decoder contribute to the memory footprint, requiring approximately 1 kB. The overall implementation requires only about 5 kB or 10 % of the stock base ROM size (44 kB). In our experiments, we were able to alter the memory layout so that it allows the customized boot loader to be up to 53 kB, without breaking the boot phase. Hence, the memory footprint of our scheme is within the limitations imposed by the PandaBoard platform. In addition, the generated Helper Data is 982 B in total, which is stored in off-chip NVM. The Fuzzy Extractor uses 982 B of SRAM start-up values to create a stable key, which constitute approximately 2 % of the available on-chip SRAM of the PandaBoard.

5.4.6. Runtime

The runtime results listed in Table 5.1 clearly show that decryption of the `u-boot.img` is the most time-consuming part of our scheme, taking about 7 s. While this overhead seems very large, note however, that for smaller device classes (i.e., class-1 device types), the image, which is to be encrypted typically is orders of magnitude smaller compared to the u-boot implementation used in our scenario. Furthermore, even in the test case at hand, several seconds can be an acceptable runtime overhead during boot, as for many embedded devices, especially those used in industrial settings, reboots are issued only rarely. Moreover, several lightweight block ciphers were proposed that observe better runtime performance compared to AES [EK07], particularly HIGHT [Hon+06] and XTEA [NW97].

5.5. Chapter Summary

In this chapter we proposed an anti-counterfeiting architecture and implementation for low-cost devices, which securely bootstraps a given firmware binary if the underlying hardware platform is legit. In particular, the presented solutions rely on intrinsic memory-based PUF instances extracted from on-chip SRAM modules, which can be found on Commercial Off-The-Shelf (COTS) MCUs. By deriving a device-specific cryptographic key from the PUF instance, using a modified boot loader, in order to decrypt the subsequent firmware application, we can effectively bind the firmware to a particular device.

Using standard hardware components to establish a hardware-based trust anchor that is used for key derivation, our approach does not require additional hardware and thus displays a software-only solution that meets the main goals of this thesis. Hence, our secure boot scheme and can be implemented leveraging the on-chip SRAM memory as a PUFs and modifying the boot loader to extract a device-specific cryptographic key to decrypt the firmware with the device-specific key. Furthermore, our solution merely requires an ephemeral key, which is physically existent only for a short period during boot time, which greatly improves resilience against key extraction attempts, compared to traditional approaches to store long-term keys in non-volatile memory.

In this chapter we showed that commercial boot loader implementations of embedded platforms, especially the PandaBoard, can be customized in such a way that makes measurement of the intrinsic SRAM-based PUF feasible. We further proved the applicability of the proposed secure boot scheme to embedded devices. While this solution protects the firmware application from extraction or downgrading and the initial phase of loading it, an approach to protect the integrity of the firmware application during its execution, taking a more powerful adversary into account, is presented in the following chapter.

Chapter 6

Hardware-Software Binding

Published Content

- Florian Kohnhäuser, André Schaller, and Stefan Katzenbeisser. *PUF-Based Software Protection for Low-End Embedded Devices.* In: *Trust and Trustworthy Computing.* 2015, pp. 3–21.

Protecting firmware applications on embedded platforms is an elaborate challenge due to the omnipresence of those devices and hence their physical exposure, also towards malicious parties. Moreover, the absence of secure hardware on most of such devices as shown in Chapter 2.2.3 leaves many of the deployed devices without any security facilities at all. In the previous chapter, a solution was presented that protects the static firmware application from extraction, and thus from a potentially malicious access to stored secrets and intellectual property, as well as its replacement. However, in order to provide a more holistic approach to establish trust in the firmware and hence the entire embedded device, the firmware execution must be protected as well.

Accordingly, a more powerful attacker must be considered, which is able to interfere with the firmware in a more dynamic way, namely during its execution on the device. The adversary's motivation remains fundamentally similar to those presented in Chapter 5, i.e., illegitimate reproduction of embedded systems, where an adversary reproduces existing devices by copying their firmware to counterfeit, cheaper hardware. This time however and given him extended rogue capabilities, the adversary may also pursue more advanced objectives. As an example, the attacker's mobile device may contain an application which requires a license. Instead of purchasing a license, the attacker bypasses the license check by manipulating the software. A further scenario is the play-back of Digital Rights Management (DRM) protected media on hardware media players, such as TV streaming devices. An attacker might insert code in the decryption function of the DRM player to intercept and extract the decrypted media. A famous example is the bypass of the DVD DRM encryption system CSS [Wik17].

In order to protect against such elaborate attacks, the execution of the software must both be tied to a particular device and be secured against manipulations. To realize an effective hardware-software binding, hardware support is required. With hardware support, the security of a protected program rests on a secret, e.g., a cryptographic key or a piece of code implemented in a physical module. Prominent examples are the Trusted Platform Module (TPM), USB dongles, and cryptographic co-processors. Nevertheless, integrated circuits dedicated to security are complex in their design, provoke deployment issues, occupy additional space on the underlying board, and lead to higher production costs. For this reason, especially legacy or low-end embedded devices lack hardware security mechanisms. However, as these devices are widely deployed and increasingly become the target of attacks [Sch14a], there is the need for a security solution that requires no specifically designed hardware.

In this chapter we explore a novel software protection approach, which is particularly suited for embedded devices. Our approach combines and extends a self-checksumming code technique [Hor+02] with Static Random-Access Memory (SRAM) Physically Unclonable Functions (PUFs) in commodity hardware to protect a firmware application against modifications and to tie its execution to a dedicated device. The integrity of the firmware is monitored by a so-called check function that computes hash values of sections of the firmware's text segment. A second check function verifies against the correctness of the PUF-based bitstream values and hence authenticates the underlying device. In turn, if response functions act on potential firmware modifications and initiate erroneous firmware behavior to prevent the firmware from further execution. Due to the usage of an intrinsic PUF as a secure key storage, our approach does not require any hardware modifications and thus can be easily retrofitted to already deployed devices. Furthermore, relying on a PUF significantly decreases the attack surface, as the secret is stored involving the PUF's physical properties. This makes physical attacks much more complicated compared to solutions based on non-volatile memory [Arm+10]. Moreover, common hardware requirements, in particular the on-chip SRAM PUF, which are the same ones discussed in Chapter 5, make this solution compatible to the secure boot scheme. Hence, it can be seen as an extension to the secure boot solution. In order to explore the applicability of our solution, we implemented the proposed scheme on a class-1 ARM Cortex microcontroller. Various security parameters allow for a balancing between security and performance. Finally, a security and performance evaluation reveals that we achieve a substantial level of security with a performance penalty of 10 %.

6.1. Architecture

Our software protection solution consists of four basic mechanisms: two *check* and two *response* functions. Leveraging two different check functions, we achieve the twofold goal of protecting the firmware integrity and binding it to a given device simultaneously. Regarding the check functions, we propose two instantiations. However, check functions can be instantiated in various other ways, depending on the firmware characteristics and the underlying hardware. Check functions measure the authenticity of the device and the integrity of the program. Response functions read these measurements, decide whether they indicate a healthy or a manipulated state, and initiate a program misbehavior if a manipulation has been detected. In order to protect a software with our protection scheme, both functions are repeatedly integrated into the software's program code.

In more detail, the first check function measures the integrity of the software by hashing its native program code (cf. Section 6.2). The second check function computes a unique bitstream on the basis of a device-dependent SRAMs PUF response to measure the authenticity of the device (cf. Section 6.3). If those two measurements indicate a manipulated state, the first response function redirects branches to random locations in the program text segment and the second response function corrupts the program's execution stack (cf. Section 6.4). Hence, if the program or the execution environment has been manipulated, both response functions cause a malfunction of the program.

6.2. Code Integrity Checks

The integrity of the executable is measured by multiple self-checksumming code segments at runtime. Each segment consists of a hash function which computes a hash value over a predefined section in the program's text segment. The hash value represents the integrity status of the checked section. It is later used by response functions to decide whether the program has been tampered with. Depending on the spatial separation of the hash function and the response function, a hash value is either stored in a register or on the stack.

In order to provide a high level of firmware protection, our proposed solution relies on both, security at the core of the architecture as well as added methods for software obfuscation [NC09]. In order to realize obfuscation and security, each hash function is inlined in the code, preferably with some spatial separation from other hash functions, and gets executed as the control flow passes the code location where the hash function is inserted. It is desirable that each inserted hash function is executed at least once at runtime, but not so frequently that the protected program suffers from a huge runtime overhead. In practice, profiling tools can be utilized to identify suitable code locations. We propose to let multiple hash functions measure a contiguous and relatively small part of the program. Thus, each integrity measurement only consumes little time. In addition, the effort for an attacker to remove the software protection increases.

In order to increase the effort even more, each code segment is measured multiple times by different hash functions. The so-called *overlap factor* indicates how often a code section is checked by different hash functions. Its value must be well-chosen to achieve a balance between security and performance according to the application scenario. To avoid that hash functions suspiciously measure large parts of the program, we recommend to split the program code in sections of equal size. These code regions are then uniformly assigned to hash functions till the overlap factor for each code region is saturated.

The design of our hash function is based on the work by Horne et al. [Hor+02]. With $d = \{d_1, ..., d_n\}$ being data in a code section which is protected by a hash function $\mathbf{H}(\cdot)$, c being an odd multiplier constant, and $h_i(d)$ being the hash value in iteration i, our hash function can formally be defined as:

$$h_i(d) = \begin{cases} 0, & i = 1 \\ h_{i-1}(d) + c \cdot d_i. & 1 < i \leq n\,. \end{cases} \tag{6.1}$$

We deviated from Horne's approach by not multiplying $h_{i-1}(d)$ with c in each iteration. This allows us to construct arbitrary complex mutually checking code regions (cf. Section 6.5). One reason we build on the code integrity check by Horne et al. is the hash function's size and speed. A large and slow hash function would fairly expand program size as well as runtime overhead, since the hash function is inlined frequently into the original program. However, the most important reason is stealth. An attacker who can locate all hash functions is able to break the code integrity check, for instance, by overwriting hash functions with code that always writes the respective expected hash value in memory. In order to mitigate this attack vector, the proposed hash function is constructed to neither contain any suspicious operations nor to provide any characteristic pattern as shown below. In addition, its implementation in native program code can easily be diversified. Thus, each hash function can be customized, leaving the attacker with no weak point for pattern matching attacks (cf. Section 6.7.1).

In order to customize hash functions, the odd constant c can be randomized, the addition can be replaced by a subtraction or an XOR operation, or a further constant can be added or subtracted after the multiplication with c. In addition, the hash function's implementation in native program code can be diversified, among others, by permuting the instruction order, permuting the assignment of variables to CPU registers, or diversifying particular instructions. A further possibility is to split the hash function code into multiple segments which are inserted with spatial separation in the original program code. With these techniques it is straightforward to generate multiple million different variations of the hash function.

Another attack vector is the code read operation performed by the hash function. It allows an attacker to find the location of hash functions by searching the code for addresses within the text segment, or by observing if and where certain registers obtain values within the text segment at runtime. To mitigate this threat, we propose to implement Horne's memory access obfuscation approach [Hor+02] which uses an additional offset when addressing data in the program text segment (e.g., with the instruction `LDR Rd, [Rn, Rm]` on ARM-based platforms). In this way, text section addresses neither appear in the code nor in a register at runtime.

6.3. Device Authenticity Check

In the device authenticity check mechanism, we leverage the microcontroller's SRAM PUF start-up values to compute a device-dependent bitstream. Since the SRAM PUF is unique and highly integrated in the microcontroller, the bitstream is unique for each embedded device. Hence, we utilize the PUF-generated bitstream in our response functions to authenticate the device at runtime.

The code for the bitstream generation is inserted into the device's boot loader. Hence, the bitstream is generated each time the device starts up. In particular, a Pseudo-Random Number Generator (PRNG) is applied to allow for a variable bitstream length. In this way, a trade-off between performance and security can be achieved. A larger bitstream takes more time to compute at device start-up but provides more unique values that can later be verified by response functions. Alternatively, it would be possible to gradually create the bitstream during program execution. However, as this further increases the execution overhead, we decided to pre-compute the entire bitstream in advance.

6.3.1. PRNG Bitstream Generation

Generating the PRNG bitstream comprises an enrollment and a reconstruction phase (cf. Chapter 3.1.3). The enrollment phase is performed at a trusted site, e.g., by the software integrator, and involves taking a reference PUFs measurement and equipping the device's boot loader with code and Helper Data to reconstruct a unique and reliable bitstream. During reconstruction, which is performed after deployment at user side and during device start-up, the equipped boot loader initializes our proposed scheme to protect the firmware during its execution. Thus, the actual bitstream is generated using the PUFs start-up values and additional error correction methods. To correct the raw PUFs start-up values from noise, they are processed by error correction mechanisms. For this purpose, we integrate a Fuzzy Extractor (FE) in the boot loader. The FE is based on the design by Bösch et al. [Bös+08] and is similar to the construction employed as part of the secure boot solution presented in Chapter 5.4.3. Hence, the techniques employed in this chapter to restore a predefined secret from SRAMs cells are based on this FE implementation.

The *enrollment phase* is performed during the deployment of our software protection scheme once for each device. Initially, a unique random secret S is chosen. Using the FE with a reference PUFs measurement X and the secret S as input, so-called Helper Data is generated and stored on the device. The Helper Data is required in the reconstruction phase to retrieve S from a single noisy PUFs measurement. Afterwards, the length for the PRNG bitstream is set, balancing security, speed, and storage consumption for the particular device and use case. At last, it is set at which location the bitstream is stored in memory during the reconstruction phase.

The *reconstruction phase* is executed each time the device is started. Initially, the boot loader measures and stores the noisy SRAM PUFs values X'. Next, the FE reconstructs a secret S' using the current PUF measurement X' and the stored Helper Data as input. If the PUFs measurement X' corresponds to the respective Helper Data, the reconstructed secret S' will match the original secret S. S' is then used to initialize the PRNG which finally generates a PRNG bitstream of the predefined length in memory.

6.4. Response Functions

Before a response function is inserted into the code, it is randomly selected whether the response function verifies a hash value, a value of the PRNG bitstream or both values at once. If a response function verifies a hash value, it uses the hash value of the nearest preceding hash function. This ensures that hash values are verified shortly after they are measured, thwarting code manipulations promptly after they have been detected. If a response function verifies a value of the PRNG bitstream, it uses a random preferably nonrecurring bitstream value. The basic idea is to use a unique address in each PRNG bitstream access. Thus, a single address cannot be used as an attack vector for pattern matching attacks or as a watchpoint in dynamic analyses. However, if there are fewer PRNG bitstream values than deployed response functions available, some addresses must be used multiple times.

The overall goal of our two response functions is to provoke a malfunction of the protected program if the measured code integrity or device authenticity values are invalid. We would like to point out that a malfunction of the program may lead to a damage of the machine that is controlled by the program. However, the alternative to perform a deterministic action (e.g., a controlled program shutdown) would provide an easy attack vector for the adversary. In this scenario, the adversary could simply observe where the program shutdown is initiated to locate the response functions in the code.

6.4.1. Indirect Branch Response

The indirect branch response is applicable to any branch in the program. When applied, an original branch is converted to an indirect branch whose target address is dependent on the verified values, i.e., either on a hash value, on a value of the PRNG bitstream or on both values. The exact target address of the indirect branch is determined by a computation which meets the following requirements.

The output of the computation must equal the target address of the replaced original branch if the verified values correspond to their expected values. If at least one of the verified values is corrupted, the outcome of the computation must be a random address that lies within the program text segment. The latter requirement ensures that the computed target address is always a valid instruction that can be executed. If the computation of the target address would not generate a valid address in the text segment, program manipulations would immediately cause memory access violations. This would be very suspicious and allows the attacker to easily locate the response function with backtraces.

In practice, the behavior of the indirect branch tamper response is highly dependent on the program size and the structure of the program code (e.g., the number of functions in the program). We observed, on average, about two function calls until a memory access violation occurred after the indirect branch response was executed.

As an additional requirement, the computation of the target address must be simple. In order to improve stealth, its implementation should be short and should not contain unusual instructions. To improve stealth even more, each deployment of the indirect branch response function should be customized, for instance, with the techniques presented in Chapter 6.2.

6.4.2. Stack Manipulation Response

In contrast to the indirect branch response, the stack manipulation response can be deployed at arbitrary locations in the program code. The idea behind the stack manipulation response is to corrupt the execution stack if the verified values are invalid. Hence, in case of an unauthorized modification, local variables, function arguments, register copies, return addresses and other data that lies on the stack, are altered. As a result, the program continues its execution with incorrect values.

A simple way to accomplish a modification of all values on the stack is to shift the stack pointer. Shifting the stack pointer has two benefits. First, it mixes up stack frames, which complicates backtracing the program. Second, it modifies the return address and thus provokes a program crash when the currently executed function returns. If an eventual program crash as a tamper-response is not desirable, we propose to alter values on the stack directly.

6.5. Mutually Checking Code Regions

Since the presented protection mechanisms secure the entire program code and at the same time are also part of the program code, they secure each other against modifications as well. Although this enhances the security of a protected software, it comes at the cost of emerging circular dependencies in the deployment process. These mutual dependencies occur because at some point code protection measures, consisting of a hash function and a response function which verifies the hash function's value, circularly check each other.

In the work by Horne et al. [Hor+02], code regions are assigned to hash functions in a left-to-right pass which generates no mutual dependencies. However, with this approach, the overlap factor is comparatively low at the beginning and the end of the program code. In fact, their overlap factor goes down to a factor of one in the first and last few bytes of the program code. By contrast, we propose a uniform assignment of hash functions to code regions and a subsequent solving of the upcoming circular dependencies. Thus, we can ensure a consistent overlap factor throughout the entire program code.

In order to solve circular dependencies, the first step is to transform mutually checking code regions into an equation system. For this purpose, we initially deploy all protection mechanisms into the software and build a temporary protected binary. The protected binary contains the final code, except for the response functions' *reference values* and additional *placeholder values*. We propose to insert one freely selectable 32 bit placeholder value per code integrity measure to facilitate solving the equation system. Let $d = \{d_1, \dots, d_n\}$ be a list of n 32 bit words in of a code section, which are protected by a hash function $\mathbf{H}(\cdot)$. Hence, the hash function computes to:

$$\mathbf{H}(d) \equiv \sum_{i}^{n} c \cdot d_i. \tag{6.2}$$

Considering the fact that d contains two unknown components, being the reference values and the placeholder values, we change d to $d = \{d_1, \dots, d_r, \dots, d_p, \dots, d_n\}$, with d_r being the reference value and d_p being the placeholder value. Next, we utilize the fact that the final hash values can be written as the sum of multiple data values. This allows us to construct a sum that consists of a variable term which comprises both variable values d_p and d_r and a fixed term that comprises the fixed, known code words $d_1, \dots, d_n$, i.e. without d_r and d_p. Let further c be the hash function's multiplier constant, $\mathbf{H}(\cdot)$ can be written as:

$$\mathbf{H}(d) \equiv \underbrace{c \cdot d_r + c \cdot d_p}_{l} + \underbrace{\sum_{i \neq r i \neq p}^{n} c \cdot d_i}_{r} \pmod{2^{32}}\,. \tag{6.3}$$

In this way, hash values are divided in a variable part l, containing the reference value d_r and the additional placeholder value d_p which are to be solved, and a fixed part r, containing the rest of the code segment. Since the code data d_i and the multiplier constant c are fixed after deployment, r can easily be computed. Next, reference values must be expressed in relation to hash values and PRNG bitstream values. The exact dependence between PRNG, hash, and reference value is given by the response function in which the reference value is used. Finally, these relations are combined to one linear Diophantine equation system which is then solved according to the approach of Lazebnik [Laz96]. In the next section we show that the equation system is always solvable, no matter how interdependent mutually checking code regions are.

6.6. On the Solvability of Mutually Checking Code Regions

In this section, the approach of handling mutually checking code regions (cf. Section 6.5) always provides a solution.

6.6.1. Assumptions

The approach is based on the assumption that the protected program text segment is segmented in disjoint parts, so-called code regions. Each code region contains one hash function and one response function which verifies the hash function's hash value. This assumption imposes no restrictions, as the the integrity protection uses a uniform deployment of hash functions in the program code segment and a prompt verification of hash values (cf. Section 6.1). In addition, we assume that all hash functions that measure a certain code region use the same computation to generate their hash value. This assumption causes no relevant degradation of stealth, as we stated many other ways to diversify the code of a hash function in Chapter 6.2. Furthermore, for convenience, we expect that each response function only verifies one hash value. For this purpose, the response function uses one reference value which is compared to the verified hash value directly to reveal a code manipulation (e.g., if **H**(code_region) $\neq$ ref_value) initiate_response();). If, in practice, the reference value should also be used to verify a value of the PRNG bitstream, to calculate the target address for an indirect branch, or to compute a stack pointer movement, a slightly more complex computation is required to calculate the reference value. However, this computation consists of just a few extra additions or subtractions and thus presents no obstacle.

6.6.2. Proof Sketch

First, we show that the additional placeholder value in each code region (cf. Section 6.5) can be used to let each code region hash to a specific hash value. With 32 bit as the target platform register length, let $d = [d_1, ..., d_r, ..., d_p, ..., d_n]$ be a list of n 32 bit words in a code section to be hashed. Let further d_r be a reference value, d_p an unknown placeholder value, and c be an odd multiplier constant. Applying the partitioning of the summed hash values into a variable and a fixed term (cf. Equation (6.3)), the hash value of the code section can formally be written as:

$$\mathbf{H}(\text{code\_region}) \equiv c \cdot d_r + \underbrace{\sum_{\substack{i \neq r \\ i \neq p}}^{n} c \cdot d_i}_{\text{fixed}} + c \cdot d_p \pmod{2^{32}} . \tag{6.4}$$

For simplicity, we now assume that we already decided for an odd constant c and somehow know the correct reference value d_r. Thus, $fixed$ can be pre-computed and we get:

$$c \cdot d_p \equiv \mathbf{H}(\text{code\_region}) - \text{fixed} \pmod{2^{32}} . \tag{6.5}$$

In general, a modular linear equation $ax \equiv b \pmod{n}$ is solvable if $g|b$, where $g = \gcd(a, n) = ax' + ny'$ [Cor09]. If $g|b$, then the equation has g solutions:

$$\begin{aligned} x_0 &= x'(b/g) \mod n \\ x_i &= x_0 + i(n/g), \quad \text{where } i = 1, 2, ..., g-1. \end{aligned} \tag{6.6}$$

Applied to Equation (6.5), there exists a solution for d_p, if $g|\mathsf{fixed}$, where $g = \gcd(c, 2^{32})$. Since c is odd, g is always 1 and the equation is solvable for an arbitrary value fixed. This implies that the placeholder value can be used to adjust the hash value of a code region to take any value. Since we required all hash functions which measure the same code region to use the very same computation to generate their hash value, all present hash functions calculate the same specified value for the adjusted code region.

We can exploit this fact to establish correct reference values. A simple solution is to set all reference values to zero and adjust all placeholder values in a way that they let their respective code regions hash to zero. Now we can make use of the hash function's property of being summarizable and commutative. Thus, as each code region hashes to zero, an arbitrary concatenation of different code regions also hashes to zero. This corresponds to the set reference value of zero. We arrive at the conclusion that the reference values match the measured hash values no matter how interdependent mutually checking code regions are.

However, in order to not always generate the same hash value in mutually checking code regions, it may be reasonable to refuse the simple solution. Instead, with the approach described in Chapter 6.5, it is also possible for many mutual dependencies to solve the equation system with a non-zero reference value.

6.7. Security Evaluation

Information security mechanisms like cryptographic primitives or secure protocols are commonly designed to be secure in the black-box model. However, we assume a much more challenging scenario where the attacker is in possession of the endpoint devices and thus has access to the implementation and power over the execution environment. This security model is referred to as white-box model [Her+09]. Taking the white-box model as a basis, we specify two attacker models, the *static attacker* and the *dynamic attacker*. We generally expect both attackers to be familiar with our software protection model, albeit we assume that they do not know the particular deployed protection code, the location of the protection code and aspects of our protection scheme which are randomized at deployment. The following sections specify the attacker models and evaluate the security of our software protection scheme against the respective model.

6.7.1. Static Attacker Model

Specification

A static attacker has the ability to perform a static analysis on a device in his possession, i.e., he can read and modify all the data stored on the device. For instance, the attacker can read and modify the content of the external memory, like the flash memory or the Dynamic Random-Access Memory (DRAM), or the internal memory, including the software with its hard-coded secrets and cryptographic keys. In addition, we presume that the static attacker can run the program and observe its input-output behavior.

The static attacker model is a reasonable assumption for an experienced attacker who lacks the ability to debug the protected program. This may be the case due to the employment of anti-debugging techniques implemented in software (e.g., the exhaustion of breakpoint registers, or the use of API functions to check if a debugger is present) or in hardware (e.g., the physical removal of debugging ports).

Evaluation

Using a disassembler, a static attacker can analyze native program code and reverse engineer the protected program. In the worst case, the attacker would comprehend the complete code and thereby know how he can circumvent our protection mechanisms. In practice, though, this task is highly laborious, as even a small program consists of a few thousand lines of machine code.

One possibility to accelerate the analysis process is to look for outstanding instructions or specific patterns in the code. In a pattern matching attack, the attacker reveals the location of the protection code by extracting a pattern from found protection mechanisms and then searching the entire program code for that pattern. Therefore, we specifically avoided the use of suspicious operations by performing short and common computations only. The implementation of our hash function requires approximately 30 B (48 B with code access obfuscation) and the response function between 12 B to 18 B. Additionally, we demonstrated in Section 6.1 that both mechanisms can easily be diversified repeatedly.

By using another technique called collusion or differential attack, an adversary compares multiple versions of a protected program to spot the location of the inserted protection mechanisms by means of their differences. The underlying idea is that different firmware instances share major common code segments. In contrast, our protection scheme likely differs between different firmware instances (due to random constants c), and hence reveals its location. In order to protect against this attack, we leverage diversification of the entire firmware application during the deployment process [Lar+14].

A very common technique applied during a static analysis is the examination of the program's execution flow. With the deployment of the indirect branch response function, branches are replaced with indirect branches whose target addresses are dependent on hash values and values of the PRNG bitstream. As both values are not known to a static analysis tool, our approach can significantly reduce the amount of useful information that an attacker can extract from a control flow analysis.

The unpredictability of the PUF-dependent PRNG bitstream in off-line attacks has an additional advantage. Since both response functions occasionally utilize a value of the PRNG bitstream for their operation, their exact behavior cannot be predicted with static analysis techniques, which are lacking knowledge about the PUF values. Thus, static analysis methods might lead to uncovering the location of the hash function which belong to a tamper-response function. However, a static attacker will not know the required PUF value, leaving him guessing the target address. This, however, is an unpromising task, especially in larger programs. As a result, it is hardly possible for a static attacker to remove the hardware-software binding.

6.7.2. Dynamic Attacker Model

Specification

A dynamic attacker inherits all abilities from the static attacker. Furthermore, the dynamic attacker has the ability to read and modify all the data on a device at runtime. With these abilities, the attacker can interrupt a program at any time, single-step through the program code, and inspect or modify memory values at runtime. Moreover, the attacker has the capability to modify a program's execution environment. He might force the program to use bogus dynamic libraries, modified operating system functionalities, or run the program in a virtual machine.

We are aware that an attacker with the stated abilities and enough resources in time and money is capable of breaking any software security mechanism. Therefore, our goal is to increase the effort for a successful attack to a level where an attack becomes uneconomical.

Evaluation

One of the most powerful debugging features when analyzing a protected program is watchpoints. Watchpoints are used to halt the execution whenever the program accesses predefined memory locations. A dynamic attacker can use this technique to locate a large fraction of all response functions by recurrently setting watchpoints on values of the PRNG bitstream while executing the program with various input. In addition, by setting watchpoints on addresses within the program text segment, the attacker can locate hash functions. A subsequent tracing of the hash functions' hash values can reveal the location of all remaining response functions. Having located all response functions, the attacker can remove the verification of hash and PRNG bitstream values and thus disable our software protection. Although the described approach is eventually successful, it requires a significant amount of effort from an attacker. Furthermore, the effort can be arbitrarily augmented by increasing the number of hash functions, setting a higher overlap factor, or obfuscating access on hash values and values of the PRNG bitstream.

Another common dynamic analysis technique is tracing. Tracing a program involves logging information during the program's execution, such as the execution path, memory values, or register values. A dynamic attacker may trace back program crashes or abnormal program behavior to localize response functions. During the design of our response functions, we ensured that there is a large spatial and temporal separation between the execution of the response function and its impact on the program, i.e., a program crash or a program misbehavior. Thus, the attacker has to examine a large portion of the trace back to finally localize a single response function.

Profiling is an additional dynamic analysis technique which involves measuring particular runtime performance values. In general, our protection mechanisms do not consume an extraordinary amount of CPU time or memory. But yet, profiling a protected program may reveal the location of deployed hash functions when the execution of a hash function takes an exceptionally long time compared to the execution time of the original program. Anyhow, profiling requires nearly the same effort as the above described approach with watchpoints, as the protected program must be examined multiple times with a different input. On top of that, the profiling approach is less reliable than the watchpoint approach, because the code that is often run through does not need to be part of a hash function.

Emulation is a further dynamic analysis technique. An emulator is a software which simulates the behavior of a particular hardware platform. With emulation, an adversary can bypass the hardware-software binding by emulating particular PUF start-up values. In addition, an adversary can redirect data access to the unmodified version and code access to the modified version of a protected program, to bypass our code integrity protection. Nevertheless, emulation attacks are unpractical, because the software has to run in an emulator and cannot run directly on the hardware of an embedded system. In addition, the performance is slower, an emulator is hard to implement, and the protected programs PRNG bitstream must be extracted.

With temporary modifications or on-the-fly writes in memory, the attacker modifies a code region before its execution and recovers it to its original form afterwards. If an adversary inserts his modification just before it is executed and restores the original code immediately after the modified code has been executed, we cannot protect against this attack. However, this requires the attacker to permanently attach a debugger to the program, to write a debugger script which performs the attack without manual intervention, and to accept a loss in performance because of multiple code manipulations at runtime.

6.8. Proof of Concept

In order to explore the applicability of our software protection scheme, we implemented and evaluated it on the Stellaris LM4F120H5QR microcontroller (cf. Section 4.2). During deployment, the protection mechanisms are inserted into the source code of the program to be protected. Subsequently, the LLVM compiler framework [LA14] with Clang front-end [LLV07] is used to compile the equipped source code to the final protected binary. For this purpose, we wrote a Python script which controls LLVM, Clang, and additional external tools and libraries to automatically build the protected program. In the following sections we give details on the implementation of the protection scheme and the performance of our implementation.

Before using a SRAM PUF instance in security critical applications, it is crucial to characterize the SRAM start-up values for constructing an efficient Fuzzy Extractor and extracting a secret key with full entropy. We refer to Section 4.5.1 for the evaluation results of the PUF characteristics of the LM4F120H5QR platform.

6.8.1. Implemented Protection Mechanisms

Hash Function

In Chapter 6.1 we stated that it is vital for the security of our protection scheme to deploy syntactically different hash functions. In order to have precise control over the hash function's native program code, we inline hash functions as ARM assembler code in the program source code. To avoid the usage of unusual instructions in our ARM assembler version of the hash function, we implemented the hash function in C and compiled it to get an assembly language prototype. An ARM assembly prototype generated in such a way is shown in Algorithm 2. It hashes a code region from address `0x26c` to `0x2ac` with the multiplier constant $c = 3$:

Algorithm 2: ARM assembly implementation of the hash function.

```
movs r1, #0                                    // hash = 0
movw r2, 0x26c                                 // start = 0x26c
movw r3, 0x2ac                                 // end = 0x2ac
loop:
ldr r4, [r2], 4                  // tmp = data[i], start++
add r4, r4, r4, LSL#1                         // tmp = 3*tmp
add r1, r4                             // hash = hash + 3*tmp
cmp r2, r3                                // if (start < end)
blt loop                                   // then goto loop
```

PRNG Bitstream Generation

During deployment, we substitute the pre-existing Stellaris boot loader with a modified version that contains our PRNG bitstream generation code. Besides the standard initialization code, the modified boot loader contains code for the extraction of the PUF start-up values, a FE based on the design by Bösch et al. [Bös+08], Helper Data to reconstruct a predefined secret, and a PRNG based on the Keccak (SHA-3) implementation of Herrewege et al. [HV14]. At first, the boot loader extracts 240 B of PUF start-up values. In a next step the original code is resumed, relocating the firmware to SRAM and executing it. Next, the FE reconstructs a predefined 128 bit secret using the PUFs start-up values and the Helper Data. Here, we reuse Keccak in the privacy amplification phase of the FE. The reconstructed secret is used to initialize the PRNG. We use Keccak as a PRNG, primarily because of its compact size and speed on ARM devices. For the length of the bitstream, we suggest to use 2×10^{17} bit bits, which provides 4096 unique values and consumes 16 kB of memory at runtime. Nevertheless, the bitstream length can be set to an arbitrary value, for instance, to consume less storage.

Indirect Branch Response Function

During deployment, existing branches in the original source code are overwritten with the code of the indirect branch response function. The target address of the indirect branch is computed by the sum of the verified values and a specific offset, modulo a unique value, plus another unique value. The following code snippet in C syntax illustrates an indirect branch to a function, which takes no argument and returns void (e.g., `void foo(void)`):

Algorithm 3: C implementation of the indirect branch function.

```
    void (*foo)(void);
    foo = ((*hash_value + *puf_prng_value + *offset);
    foo();
```

If the hash value and the PRNG value match their expected values, *offset*, *modulo*, and *shift* adjust the indirect branch to match the original target address. In order to provide no constant value as an attack vector for pattern matching attacks, *modulo* and *shift* are randomized between certain bounds in each deployment of the indirect branch response function.

Stack Manipulation Response Function

We insert each stack manipulation response function randomly between the location of the corresponding hash function and the subsequently executed hash function. Our implemented stack manipulation response function either increments or decrements the stack pointer by a random value between 4 and 24 B in case the check function indicates a modification of the firmware.

6.8.2. Performance Evaluation

Due to the lack of open source applications for the Stellaris platform, we developed our own evaluation program. The evaluation program encrypts and decrypts a 16 B string using AES 128 bit, sends the plaintext and the ciphertext to the Univeral Asynchronous Receiver Transmitter (UART) port, and measures the amount of CPU cycles consumed from the start to the end of the main function.

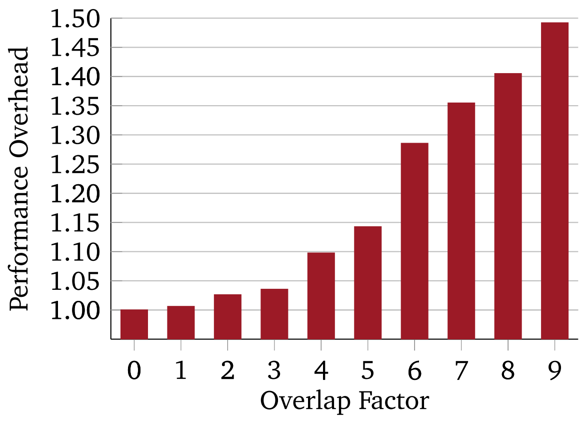


Figure 6.1.: Runtime performance comparison with different overlap factor settings.

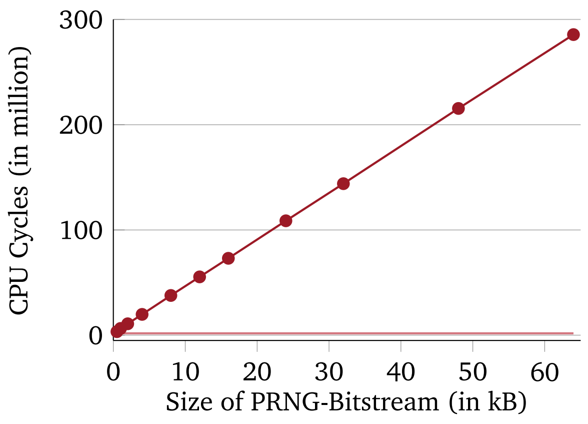


Figure 6.2.: Start-up runtime performance with varying bitstream size.

For the deployment of our software protection scheme, we used the following security settings. We inserted one code integrity check mechanism in each function of the evaluation program. As 9 of the 11 deployed functions are executed at runtime, we generate a coverage of 82 % This is a realistic scenario, as a real application will certainly contain functions that are not always executed at runtime (e.g., whose execution depends on specific user input). In addition, we used a PRNG bitstream length of 2×10^{17} bit, which corresponds to a size of 16 kB. For the deployment of the response functions, we inserted the stack manipulation response in each circular dependent code region and the indirect branch response in the remaining code.

Runtime Performance

In our runtime evaluation, we deployed the evaluation program with the above mentioned security settings and a variable overlap factor. Figure 6.1 illustrates the relative runtime overhead for various overlap factor preferences averaged over 10 measurements per overlap factor. The runtime of the original unprotected program is represented with an overlap factor of zero and an overhead factor of one. As the overlap defines how many times a code region is checked by different hash functions, an increasing overlap factor increments the amount of code lines that each hash function has to check. In our experiments using an overlap factor of nine, each hash function almost checks the complete text segment, which generates an overhead of approximately half of the original runtime. It is evident that such an overhead is not acceptable in most applications. On the other hand, even when each code region is checked by three different hash functions, the runtime overhead is below 5 %. As this slow-down will only be noticed by sensitive users, we can easily recommend an overlap factor of three for conservative usage.

Another performance overhead originates from the generation of the PRNG bitstream at device start-up. Figure 6.2 depicts the amount of CPU cycles that is required to compute a bitstream of a specific length. For comparison, the original program consumes roughly 1.8 million CPU cycles (horizontal plot at the bottom). The figure illustrates that there is almost a proportional relationship between the size of the PRNG bitstream and the amount of CPU cycles. Thus, compared with the calculation of the pseudo-random values, the extraction of the PUFs start-up values and the execution of the Fuzzy Extractor barely uses any CPU time. The figure also shows that the generation of the PRNG bitstream consumes much more CPU resources than the execution of the actual program. However, it must be considered that the PRNG bitstream is only generated at device start-up. Assuming the embedded devices are clocked at 50 MHz, a bitstream size of 16 kB delays the start of the device by 1.5 s seconds which is likely to be acceptable for most applications.

Storage Consumption

The program size overhead of a protected program is dependent on the number of inserted hash functions, the choice of the response function and the number of inserted response functions. For evaluation, we deployed our protection mechanisms using the previously mentioned security settings. In this way, we obtained a protected program which was on average 63 % larger than the equivalent unprotected program. Another storage overhead arises at runtime due to the execution of both check mechanisms. However, the memory consumption of the hash functions is negligible, as each value resides just a short time in memory and only occupies 4 B of storage. In contrast, the values of the PRNG bitstream are kept in memory permanently, and they consume 16 kB of memory for our proposed bitstream length. Nevertheless, by setting another bitstream length, the runtime memory overhead can be adjusted as required.

6.9. Chapter Summary

In this chapter, we explored a novel hardware-assisted software protection approach, which combines existing software-based techniques with PUFs. Using a microcontroller's SRAM as a PUF instance, we overcome the drawbacks of traditional hardware tamper-proofing solutions. The proposed software protection scheme ties the execution of a software instance to a specific device, protects its program code against manipulations and can easily be retrofitted to already deployed devices. To demonstrate our approach, we implemented it on a low-cost ARM-based microcontroller. By adjusting certain security parameters, it is possible to balance security with performance. We showed that our software protection scheme offers a high level of security against a static adversary and demonstrated that a dynamic adversary requires a considerable amount of resources to perform a successful attack. A further performance evaluation showed that an extensive level of security is achievable with an acceptable performance degradation of ten percent.

Chapter 7

Authentication

Published Content

- André Schaller, Wenjie Xiong, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Škorić, Stefan Katzenbeisser, and Jakub Szefer. *Decay-Based DRAM PUFs in Commodity Devices.* In: *IEEE Transactions on Dependable and Secure Computing (2017).* Submitted.

One of the fundamental means that establish trust in remote, interconnected entities is authentication. Classical authentication scenarios usually involve a two-party setting,[1] where a verifier $\mathcal{V}$ wants to get assurance about the identity of a remote prover $\mathcal{P}$, hence mitigating a possible impersonation of $\mathcal{P}$. Extending this setting so that authentication of the verifier towards the prover is required as well, leads to the notion of *mutual* authentication, which establishes trust in the authenticity of both parties, hence additionally ruling out the existence of a malicious verifier. The authors of [MVV96] distinguish three fundamental approaches to authentication, including password-based methods, protocols that rely on challenge-response mechanisms, custom constructions and zero-knowledge protocols. Hereafter, we focus on such authentication protocols that leverage an underlying challenge-response procedure.

While the focus of authentication protocols is put on proving the verifier's authentication, there are differences in the actual assurances that various authentication schemes provide. Whereas some constructions allow for authenticating individual users of a given system, others establish trust in the underlying hardware platforms. Moreover, some authentication protocols guarantee the integrity of data, which was generated by the party that is to be authenticated. There are even more subtle differences regarding the authentication guarantees among different constructions. Authentication guarantees may focus on the origin of data generation, the time of generation or the "freshness" of the received data at verifier side. Beside authenticity, some authentication protocols [NT94] also provide confidentiality of the data communicated during the protocol runs. In this way, such authentication protocols mitigate the threat of eavesdroppers intercepting the communication between prover and verifier, which is one of the major weaknesses of naive password-based authentication.

In the literature there are numerous works that present approaches to traditional authentication solutions [BAN89; BP92; NT94]. Such approaches rely on classical cryptography, i.e., symmetric or public-key cryptography and most of them are based on a challenge-response protocol that comprises a verifier $\mathcal{V}$ and a prover $\mathcal{P}$, as mentioned above. Such authentication protocols generally require the underlying hardware to possess enough computational resources to execute

[1] Some constructions require a trusted third party, which will not be considered in this chapter.

primitives of the involved cryptographic scheme and to implement secure hardware or trusted execution environments that allow for secure processing of the involved assets, such as keys, nonces and hashes.

With the emergence of low-cost devices in the context of the Radio-Frequency Identification (RFID) and Electronic Product Code (EPC) technologies however, device authentication again became a technological challenge to be solved. Such platforms mainly belong to low-cost class-0 device types and hence are highly resource-constrained. Accordingly, these hardware constraints impede the application of traditional application schemes on such devices, in fact rendering them impossible [V+03; Jue+04; Lee+05]. In order to tackle this issue, so-called lightweight authentication protocols were proposed, that try to avoid the usage of cryptographic primitives and instead leverage more lightweight approaches.

7.1. Existing Authentication Solutions

Initial efforts to lightweight authentication, which simultaneously aim at providing privacy, involved the usage of pseudonyms. Juels [Jue+04] suggested that each device should store a list of multiple, random-looking pseudonyms and to use a different one during each authentication run. The main drawback of this approach is that a Man In The Middle (MITM) attacker is able to gather all pseudonyms, at some point he is able to undermine authenticity as well as privacy at once. A different approach that involves computing hashes over keys is denoted as “hash-lock” authentication [SWE02]. Here, the prover computes a hash over a device-unique key before exposing its ID to the verifier. This hash-based approach led to a series of similar authentication schemes, including the Randomized Hash Lock [Wei+04] and the hash-based ID variation scheme [HM04]. While most of these protocols provide data privacy during authentication, all of them suffer from not being trivially implementable on resource-constrained devices. The usage of a cryptographic hash function that is a fundamental principle of these protocols, impedes their straight forward application on class-0 RFID devices. Even more computationally expensive cryptographic primitives are used by authentication protocols that rely on re-encryption [JP03; SRS04]. The basic idea here is to periodically re-encrypt the already encrypted information, stored on the device. Although they provide stronger security as these methods predominantly rely on public-key cryptography[2], it is obvious that they are not usable on low-cost devices, given their rigorous hardware restrictions. A supposedly more lightweight approach to authentication was proposed by Hopper and Blum, who facilitated human-computer authentication by exploiting the hardness of the learning parity with noise problem [HB01]. This approach led to a series of derivatives, which resembles the so-called family of *HB protocols* [J+05; BCD06; MP07]. One central drawback of the HB protocols is their susceptibility to MITM attacks. In particular, the original HB protocol [HB01], as well sits predecessor HB^{+} are not secure in the active adversary model, as was shown in [GRS05]. Various attempts to mitigate the central problem of vulnerability against MITM attacks and efforts to further improve efficiency lead to a series of derivatives. The intent of HB^{++} [BCD06] was to be secure in the active adversary model. Similarly, HB-MP [MP07] tried to achieve the same goal and to achieve increased performance at the same time. However, the authors of [GRS08] have proven its insecurity for certain MITM attacks. Moreover, Armknecht [AHM14] challenged the efficiency claims of the HB protocol family, which were designed to be applicable to highly resource-constrained platforms in the first place. Identifying realistic constraints of highly resource-constrained RFID tokens, Armknecht concluded that

[2] Some of them leverage symmetric cryptography, which however, leads to expensive multiple decryption rounds.

none of the existing lightweight HB authentication protocols, which rely on the LPN problem, are applicable to those devices. Subsequently, protocols have been proposed that allow for mutual authentication, which are tailored to usage on highly resource-constrained devices and were hence dubbed ultra lightweight mutual authentication protocols (UMAP) [Per+06a; Per+06b; Chi07]. With the emergence of Physically Unclonable Functions (PUFs), the need for lightweight attestation was once again stimulated as PUF-enabled devices usually exhibit constraints similar to the class of RFID token, with respect to their hardware capabilities. However, the additional and implicit need for error-correction of PUF-based protocols led to a series of new authentication protocols specifically tailored to PUFs usage [Gas+02; BR07; ÖHS08; SVW10; Kat+11].

7.2. Mutual Authentication Protocol

In this chapter, we present a novel lightweight protocol that allows for mutual device authentication between two remote parties. In particular, we assume the prover $\mathcal{P}$ to be a resource-constrained device that implements the decay-based Dynamic Random-Access Memory (DRAM) PUF, presented in Section 3.3.1, whereas the second party possesses more computational capabilities. The protocol is tailored to the characteristics of the decay-based DRAM PUF and is particularly lightweight as it does not require the usage of costly error-correcting codes as part of Fuzzy Extractor constructions.

If a device supports the computation of Helper Data, i.e., by applying a Fuzzy Extractor, it can immediately provide stable PUF keys for use in symmetric or asymmetric cryptographic protocols. However, in this chapter we consider the case of a highly resource-constrained device which does implement DRAM, but which *does not* possess the processing power to run the key reconstruction phase of the Helper Data System. Especially the `FE.Rec()` function is computation-intensive as it either requires a large number of gates or results in long execution times [Van+12].

In this lightweight scenario, the PUF device is also unable to perform any cryptographic operation. Hence, if we want to construct an authentication scheme based on PUF responses, then the parties will inevitably have to transmit information about PUF responses *in plaintext*. This makes most lightweight protocols susceptible to MITM attacks. Nevertheless, the development and implementation of lightweight authentication is a meaningful task, as it presents a cost-effective hurdle against 'casual' attacks. Such attacks resemble the main attack vector in the face of low-cost devices that usually implement no security protocols whatsoever. Respective attacks are covered by the Dolev-Yao model, where the adversary can eavesdrop, inject, alter or suppress messages sent over the communication channel [DY83].

In order to challenge such casual threats, we present a lightweight PUF-based mutual authentication protocol for this scenario. Here, a prover device $\mathcal{P}$ needs access to some resource offered by a verifier $\mathcal{V}$, and has to prove that it possesses a specific logical PUF $\mathbf{PUF}_{id}$. Furthermore, $\mathcal{P}$ trusts the resource only if $\mathcal{V}$ too knows the responses of $\mathbf{PUF}_{id}$. In particular, the protocol is based on the mutual comparison of sets s_x which contain indices of decayed DRAM cells at increasing time scales of decay times $\mathtt{t}_x$. The prover $\mathcal{P}$, in possession of the PUF, reveals the set s_x to the verifier $\mathcal{V}$, which is randomly contaminated with indices pointing to undecayed DRAM cells. The verifier $\mathcal{V}$, which stores a priori PUF measurements, in turn demonstrates its ability to identify which indices belong to s_x, therefore realizing mutual authentication of $\mathcal{P}$ and $\mathcal{V}$.

7.2.1. Threat Model

The attacker model considers an adversary $\mathcal{A}$ who is able to observe the communication between $\mathcal{P}$ and $\mathcal{V}$, and also to engage in a protocol exchange with either $\mathcal{P}$ or $\mathcal{V}$. However, MITM attacks or message modifications are out of scope of the model, as such attacks are generally hard to mitigate in the context of lightweight protocols as shown above. Moreover, the protocol is publicly known, including all the system parameters.

Consider an active adversary whose aim is to obtain PUF responses by pretending to be one of the parties. The proposed scheme is built so that it displays the following two main properties: i) If the attacker impersonates the PUF device $\mathcal{P}$, the protocol should force him to be the first party to provide information about the PUF response. Thus, the attacker does not easily get access to the resources that $\mathcal{V}$ is protecting; i.e., the attacker first needs to learn PUF responses. ii) Secondly, if on the other hand the attacker impersonates $\mathcal{V}$, it should not be easy for him to *quickly* extract all the PUF responses from $\mathcal{P}$. In order to satisfy this requirement, we make sure that the initiative to start the protocol lies with $\mathcal{P}$. In this way the attacker has to wait until $\mathcal{P}$ initiates contact.

7.2.2. System Setup

Let l_{t_x} denote the number of flipped bits at decay time t_x, i.e., $l_{\mathrm{t}_x} = |s(\mathrm{t}_x)|$. At the verifier side, a set of enrollment times $\mathcal{T}_{enroll} = \{\mathrm{t}^e_0, \mathrm{t}^e_1, ..., \mathrm{t}^e_n\}$ is chosen to evaluate the bit error rates of the given DRAM PUF, which are used to set system parameters Δ_1 and Δ_2 accordingly. In particular, both thresholds are set as $\Delta_1 = \mathrm{BER}_1$ and $\Delta_2 = \max(\mathrm{BER}_1, \mathrm{BER}_2)$. Both bit error rates BER_1 and BER_2 are calculated according to Equation (7.1) and (7.2), which are explained in more detail in Section 7.3.

Similarly, the prover carefully choses a vector $\mathcal{T} = \{\mathrm{t}_0, \mathrm{t}_1, ..., \mathrm{t}_n\}$ of decay times (with $\mathrm{t}_0 < \mathrm{t}_1 < ... < \mathrm{t}_n$) so that $\forall_x \; l_{\mathrm{t}_{x+1}} - l_{\mathrm{t}_x} = \epsilon_{\mathrm{t}_x}$, i.e., at every time step the number of newly decayed cells always equals the security parameter ϵ_{t} (cf. Section 4.5.2). For the sake of clarity, we set $l_{\mathrm{t}_{x+1}} - l_{\mathrm{t}_x} = \epsilon_{\mathrm{t}_x}$. However, in practice, the number of newly decayed cells does not have to be exactly ϵ_{t} but can be similar. Due to the high number of available DRAM cells, this does not affect the security of the protocols.

7.2.3. Enrollment Phase

Enrollment is conducted by a trusted party $\mathcal{SYS}$, such as a manufacturer or a system integrator. $\mathcal{SYS}$ queries the PUF at decay times $\mathcal{T}_{enroll}$ and gets a set of measurements for each $\mathbf{PUF}_{id}$: $\mathcal{M}_{id} = \{s^{id}_e(\mathrm{t}_0), s^{id}_e(\mathrm{t}_1), \ldots, s^{id}_e(\mathrm{t}_n)\}$. For each $\mathbf{PUF}_{id}$ the prover device initializes the counter c_{id} to zero, and the verifier initializes the counter c'_{id} to zero.

7.2.4. Authentication Phase

Algorithm 4 specifies the authentication phase of the protocol. In step 2 the process of identification is conducted by sending a public identifier id to $\mathcal{V}$, which is usually stored in Non-Volatile Memory (NVM) on $\mathcal{P}$. After measuring the DRAM PUF in step 3, $\mathcal{P}$ computes a set $\mathcal{B}$ in step 4. The selection of the random set $\mathcal{B}$ ensures that the protocol is not hindered by the potentially

Algorithm 4: Mutual Authentication

```
Let $\mathcal{N}_{id}$ denote the set of the entire addresses of all the DRAM cells in $\mathbf{PUF}_{id}$.
1  $\mathcal{P}$ initiates contact.
2  $\mathcal{V}$ sends $id$ to $\mathcal{P}$.
$\mathcal{P}$ performs the following actions:
3  Set $x = c_{id}$ and perform a measurement of $\mathbf{PUF}_{id}$ at decay time $\mathrm{t}_x$, resulting in a set of
   addresses $s^{id}(\mathrm{t}_x)$ of decayed DRAM cells.
4  Randomly select DRAM cell addresses into a set $\mathcal{B} \subset \mathcal{N}_{id} \setminus s^{id}(\mathrm{t}_x)$ of size
   $|\mathcal{B}| = 2\epsilon_\mathrm{t}(x+1) - l^{id}_{\mathrm{t}_x}$.
5  Construct a vector $z$ by randomly permuting $s^{id}(\mathrm{t}_x) \cup \mathcal{B}$.
6  Construct a bitstring $a \in \{0,1\}^{2\epsilon_\mathrm{t}(x+1)}$ such that $a_i = 1$ if $z_i \in s^{id}(\mathrm{t}_x)$ and $a_i = 0$
   otherwise.
7  Increase $c_{id}$ and send $x, z$ to $\mathcal{V}$.
$\mathcal{V}$ performs the following actions:
8  Continue only if $x \geq c'_{id}$ and $z$ has length $2\epsilon_\mathrm{t}(x+1)$; else abort.
9  Construct $a' \in \{0,1\}^{2\epsilon_\mathrm{t}(x+1)}$ such that $a'_i = 1$ if $z_i \in s^{id}_e(\mathrm{t}_x)$.
10 If the fractional Hamming weight of $a'$ is larger than $\frac{1}{2}(1-\Delta_1)$, $\mathcal{P}$ is authenticated, set
   $c'_{id} = x+1$ and send $a'$; else abort.
$\mathcal{P}$ performs the following actions:
11 $\mathcal{P}$ checks if the fractional Hamming distance between $a$ and $a'$ is smaller than $\Delta_2$. If so,
   $\mathcal{V}$ is authenticated; if not, $\mathcal{P}$ aborts.
```

large number of erroneous bit flips. Although the probability of such an error per cell can be small for certain DRAM PUF instances, such as the PandaBoard (cf. Section 7.3), the huge number of involved DRAM cells in a logical PUF may drive up the total number of bit errors. Note that $\mathcal{P}$ adjusts the size of $\mathcal{B}$ so that the bitstring z, which is computed subsequently, has size $2\epsilon_\mathrm{t}(x+1)$.

In step 5 of the authentication phase, the prover constructs said address vector z from the addresses $s^{id}(\mathrm{t}_x)$ of decayed DRAM cells and a random set $\mathcal{B}$ of addresses that have not yet decayed. The random permutation in step 5 ensures that attackers cannot derive $s^{id}(\mathrm{t}_x)$, i.e., the actual PUF measurement, from z. Due to the tuning of the string length, the string a, computed in step 6, is balanced, i.e., it contains approximately as many '0's as '1's, ensuring large entropy of the bitstring a given z. Given that $\mathcal{P}$ has knowledge of the temperature behavior of his DRAM PUF, he can use a temperature-scaled decay time t' (cf. Equation 4.6) in order to retrieve z.

Letting multiple instances of the protocol run, we assume that the attacker knows the locations of flipped bits at previous decay times. In particular, an eavesdropper Eve knows $l^{id}_{t_{x-1}} \approx \epsilon_\mathrm{t} \cdot x$ addresses that also appear in $s^{id}(\mathrm{t}_x)$. Hence, the number of addresses unknown to Eve is $\approx \epsilon_\mathrm{t}$. The entropy of a given z is then the entropy of ϵ_t positions out of $(x+1)2\epsilon_\mathrm{t} - x\epsilon_\mathrm{t}$, i.e., $\log\binom{(x+2)\epsilon_\mathrm{t}}{\epsilon_\mathrm{t}}$, which can be approximated as $\epsilon_\mathrm{t}(x+2)h(\frac{1}{x+2}) \geq \epsilon_\mathrm{t}$.

Note that $\mathcal{P}$ keeps track of c_{id}, otherwise an attacker could impersonate a verifier and learn the complete memory state for each id, t_x by communicating with $\mathcal{P}$ many times. Furthermore, $\mathcal{V}$

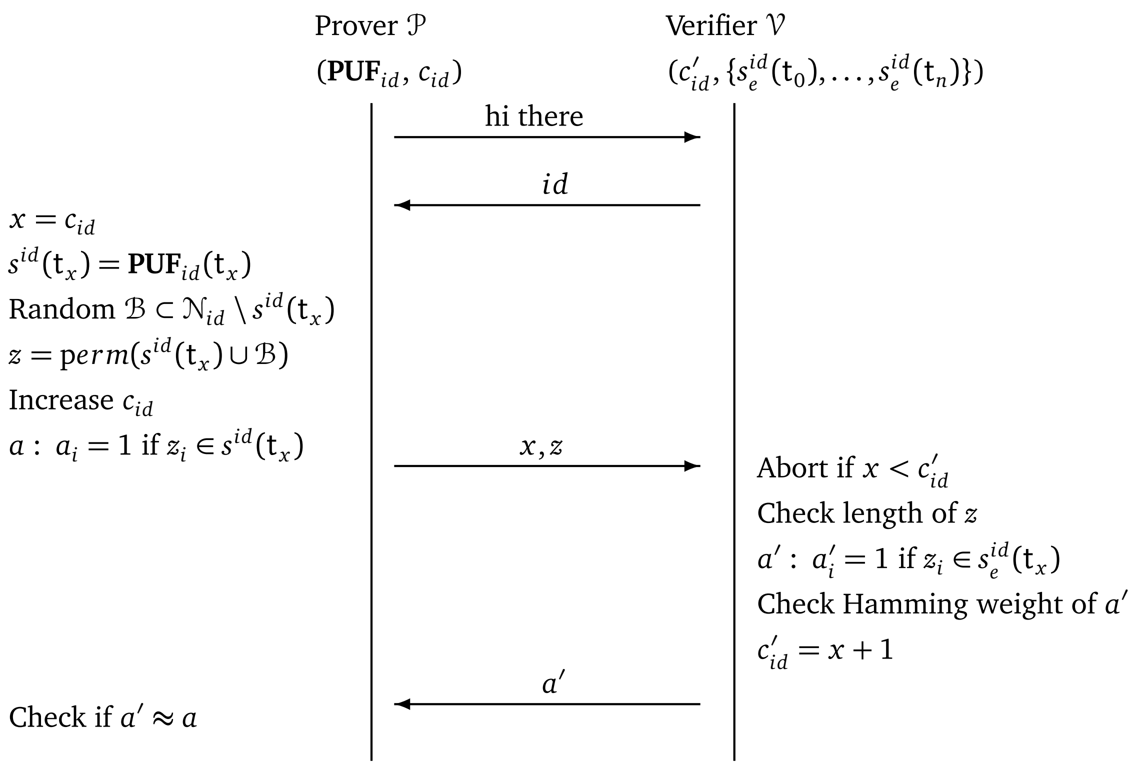

Prover $\mathcal{P}$
$(\mathbf{PUF}_{id}, c_{id})$
Verifier $\mathcal{V}$
$(c'_{id}, \{s_e^{id}(\mathrm{t}_0), \ldots, s_e^{id}(\mathrm{t}_n)\})$
hi there
id
$x = c_{id}$
$s^{id}(\mathrm{t}_x) = \mathbf{PUF}_{id}(\mathrm{t}_x)$
Random $\mathcal{B} \subset \mathcal{N}_{id} \setminus s^{id}(\mathrm{t}_x)$
$z = perm(s^{id}(\mathrm{t}_x) \cup \mathcal{B})$
Increase c_{id}
$a: \ a_i = 1$ if $z_i \in s^{id}(\mathrm{t}_x)$
x, z
Abort if $x < c'_{id}$
Check length of z
$a': \ a'_i = 1$ if $z_i \in s_e^{id}(\mathrm{t}_x)$
Check Hamming weight of a'
$c'_{id} = x + 1$
a'
Check if $a' \approx a$

Figure 7.1.: The lightweight mutual authentication protocol.

also has to keep track of c'_{id}, otherwise an attacker could replay a z from the past. The check if $x \le c'_{id}$ is meant to detect replays. In step 10 the verifier performs a check on the Hamming weight of a'. This verifies if $\mathcal{P}$ is authentic. If z is sent by an impostor, then with a very high probability z will not contain $\epsilon_{\mathrm{t}}(1 - \Delta_1)$ addresses that are also in the enrollment $s_e^{id}(\mathrm{t}_x)$.

The prover device $\mathcal{P}$ uses every challenge $(\mathbf{PUF}_{id}, \mathrm{t}_x)$ (cf. Section 3.3.1) only once. As soon as $\mathcal{P}$ sends a string z, it increases its counter c_{id}. This is independent of the a vs. a' verification at the verifier side. Note that an attacker can pretend to be $\mathcal{P}$ and make many attempts to authenticate to $\mathcal{V}$ without affecting c'_{id}.

Note that there is a straightforward denial of service attack. The attacker can repeatedly pretend to be a verifier and abort at step 8 of the protocol. With each aborted run, $\mathcal{P}$ is forced to increase the counter x. Subsequently, $\mathbf{PUF}_{id}$ will be exhausted at some point. However, given the vast amount of DRAM cells typically available (e.g., 1 GB on the PandaBoard), it is possible to instantiate thousands of logical PUF instances (cf. Section 4.5.2) on a single device. Furthermore, as $\mathcal{P}$ is the party that initiates the protocol, the attacker cannot set the pace of his denial of service attack. Lastly, $\mathcal{P}$ can be programmed to (temporarily) stop communicating if it observes consecutive failures. A sequence diagram of the protocol is depicted in Figure 7.1.

7.3. Experimental Validation

In order to find realistic reference values for the threshold parameters Δ_1 and Δ_2, which are used during the setup phase of the authentication protocol, we analyzed measurements obtained from the two device types, employed as platforms to evaluate the decay-based DRAM PUF (cf. Section 4.5.2), i.e., the PandaBoard and the Intel Galileo. In particular, we estimated the noise, by means of computing two fractional Bit Error Rates (BER) as shown below.

As discussed in Section 7.2.2, the authentication protocol uses two sets of decay times, namely $\mathcal{T}_{enroll} = \{\mathrm{t}_0^e, \mathrm{t}_1^e, ..., \mathrm{t}_n^e\}$ set by the verifier and $\mathcal{T} = \{\mathrm{t}_0, \mathrm{t}_1, ..., \mathrm{t}_n\}$ chosen by the prover accordingly. During the authentication phase the prover $\mathcal{P}$ measures the PUF using the next unused decay time $\mathrm{t}_x \in \mathcal{T}$, according to counter c_{id}. The verifier $\mathcal{V}$ uses the corresponding enrollment decay time $\mathrm{t}_x^e \in \mathcal{T}_{enroll}$ based on c'_{id}, in order to verify the prover's authenticity. Given the two set of measurements taken by the prover and verifier respectively, bit errors might occur, even in the case that both sets of decay times are identical, i.e., $\mathcal{T}_{enroll} = \mathcal{T}$. In particular, the following two bit errors have to be considered.

The first error rate BER_1 depicts the proportion of cells, which decayed during the authentication phase at decay time t_x but not at the corresponding enrollment decay time t_x^e, normalized by the total amount of cells, which decayed until t_x:

$$\mathrm{BER}_1 = \frac{|s(\mathrm{t}_x) \setminus s(\mathrm{t}_x^e)|}{l_{\mathrm{t}_x}}. \tag{7.1}$$

In contrast, BER_2 represents the proportion of such DRAM cells, which decayed at a given enrollment decay time t_x^e but did not decay during the authentication phase at the corresponding decay time t_x, normalized by the number of all the cells that comprise a measurement, except those flipped at t_x. BER_2 is computed as:

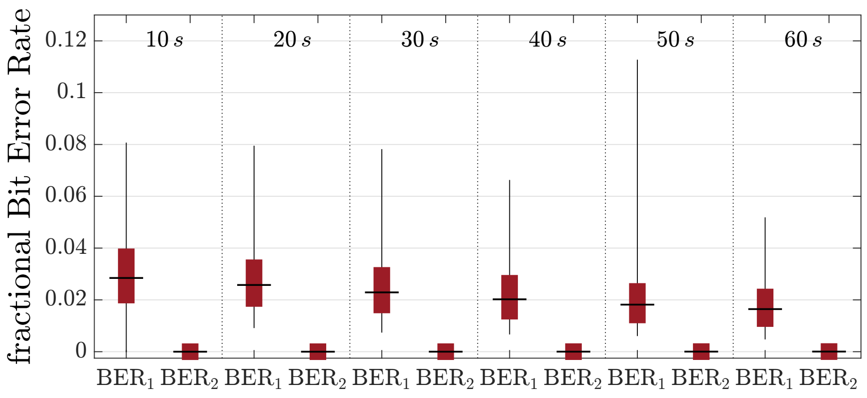
$$\mathrm{BER}_2 = \frac{|s(\mathrm{t}_x^e) \setminus s(\mathrm{t}_x)|}{\mathcal{N} - l_{\mathrm{t}_x}}. \tag{7.2}$$

Hence, BER_2 depicts the error of falsely identifying cells as not flipped during authentication, in order to construct $\mathcal{B}$, although they have been identified to be flipped during enrollment.

For evaluation, we considered the case of $\mathcal{T}_{enroll} = \mathcal{T}$ and compared pairs of measurements, computing both bit error rates. In particular, we set $\mathcal{T}_{enroll} = \mathcal{T} = \{10\,\mathrm{s}, 20\,\mathrm{s}, 30\,\mathrm{s}, 40\,\mathrm{s}, 50\,\mathrm{s}, 60\,\mathrm{s}\}$ and `size` $= 16\,\mathrm{MB}$ as done during the initial evaluation of the DRAM PUF (cf. Section 4.5.2). Figures 7.2 show the fractional bit error rates of both device types as boxplots. The figures indicate that the BER_1 dominates the overall bit error rate. They show a maximum of ≈ 0.12 for the PandaBoard and 0.55 for the Intel Galileo. Obviously, the maximum BER_2 is negligible, as only about 0.0016 bits erroneously flip on the PandaBoard and less then 0.0001 on the Intel Galileo. While the BER_1 bit error rates of the PandaBoard are comparably small, the BERs of the Intel Galileo are up to 5 times higher.

Overall, the BERs, which are used as threshold parameters to determine authenticity of $\mathcal{P}$ and $\mathcal{V}$, are below the noise levels of both device types (cf. Chapter 4.5.2). Hence, the limited erroneous characteristics allow for constructing and incorporating set $\mathcal{B}$[3] of random, undecayed DRAM cells to construct z. Thus, by using more DRAM cells as input to the protocol (i.e., undecayed cells that constitute $\mathcal{B}$), the influence of bit errors can be reduced by a factor of two, as a and a' consist of approximately the same number of decayed and undecayed cells. Given the vast

[3] Note that the size of $\mathcal{B}$ is determined by ϵ_{t}, which in turn is linked to the noise levels.



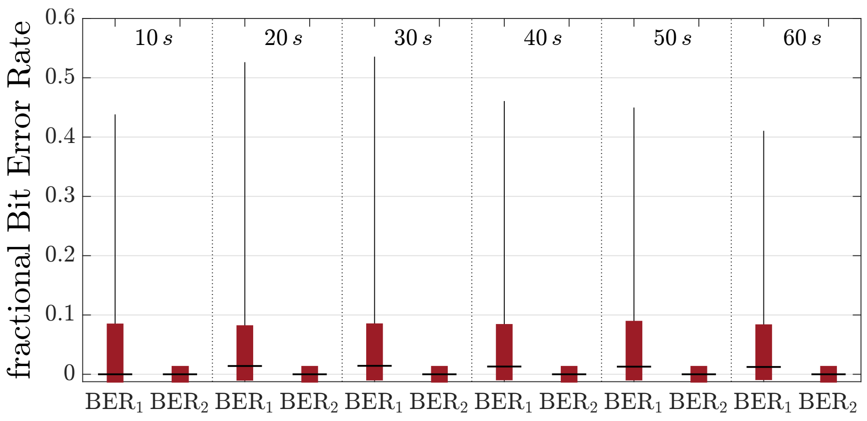


Figure 7.2.: Bit error rates BER_1 and BER_2 of the PandaBoard (top) and Intel Galileo (bottom).

number of available DRAM cells, even in the presence of high bit error rates, it is possible to find useful thresholds that allow for robust authentication of the protocol.

7.4. Chapter Summary

In this chapter we presented a novel and lightweight mutual authentication protocol that exploits the decay-based DRAM PUF, which was presented in Chapter 3.3.1. The protocol is ultra lightweight in the sense that it does not require expensive error-correction, which usually is done during the reconstruction phase of the Fuzzy Extractor, to establish a secret key. Instead, the protocol relies on the comparison of sets of indices of decayed DRAM cells and corresponding threshold values in order to authenticate both the prover and the verifier.

In this way, robust authentication can be implemented on today's DRAM-enabled System on Chip (SoC) devices and further on future low-cost MCU-based platforms, which we anticipate to implement embedded DRAM modules.

Having shown that robust authentication can be realized even on highly resource constrained devices, we will dedicate the next chapter to another important means to establish trust in remote computing entities. In particular, in the next chapter we will propose a lightweight attestation scheme, that allows devices to recover from potential attacks into a secure stage and to prove their trustworthiness towards a remote verifier.

Chapter 8

Remote Attestation

Published Content

- Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. *Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors.* In: *European Symposium on Research in Computer Security.* 2017, pp. 437–455.

A major challenge in computer security is about establishing the trustworthiness of remote platforms. The need for remotely establishing trust in computing platforms is existent even in scenarios where devices are not connected to the Internet or where they are not intended to receive firmware updates at all. For instance, SD-cards and USB sticks that are exchanged with third parties can be infected or replaced by malicious hardware in order to attack the host [b313; NL14]. Bluetooth devices may offer an even larger attack surface, since typically employed security mechanisms were shown to be insufficient [Rya13; SW05]. *Remote attestation* is a key security capability in this context, as it allows a third party to identify a remote device and verify its software integrity.

Unfortunately, existing attestation solutions either provide low security, as they rely on unrealistic assumptions, or are not applicable to commodity low-cost and resource-constrained devices, as they require custom secure hardware extensions that are difficult to adopt across Internet-of-Things (IoT) vendors. In this chapter, we propose a novel remote attestation scheme, named *Boot Attestation*, that is particularly optimized for low-cost resource-constrained embedded devices. In Boot Attestation, software integrity measurements are immediately committed to during boot, thus relaxing the traditional requirement for secure storage and reporting. Our scheme is very light on cryptographic requirements and storage, allowing efficient implementations, even on the most low-end IoT platforms available today. As we will show in Section 8.5, our scheme is supported by off-the-shelf devices. To this end, we review the hardware protection capabilities for an ARM-based commercially available platform and present implementation results.

8.1. Existing Attestation Solutions

Previous work on attestation addresses *hardware-based* or *timing-based* attestation, *scalable attestation* for groups of devices, and *secure code updates*. Hardware-based attestation schemes rely on secure hardware, such as Intel SGX or a Trusted Platform Module (TPM) [Ana+13; Tru11], that is installed on the prover. Since such secure hardware is typically too expensive and complex to be integrated in low-cost embedded devices, recent works focused on the advancement of new minimalist security architectures [Eld+12; HLN17; Koe+14b; Noo+13; Fra+14], which enable hardware-based remote attestation capabilities for small embedded devices. However, these lightweight architectures have not yet reached the market, and hence are not available in commodity low-end embedded devices. Furthermore, even when they are available, there is still the need to secure old systems.

By contrast, timing-based attestation schemes do not require secure hardware and thus are applicable to legacy systems [Kov+12; LMP11; Ses+05; Ses+04]. However, they rely on assumptions that have proven to be hard to achieve in practice [Arm+13; Cas+09; Li+15]. Such assumptions include an optimal implementation and execution of the protocol, exact time measurements, and an adversary who is passive during attestation.

Recent works address a scalable attestation of groups of devices (i.e., device swarms) that are interconnected in large mesh networks [Amb+16; Aso+15; Car+17]. The basic idea is that neighboring devices mutually attest each other in order to distribute the attestation burden across the entire network. Since these works rely on hardware-based attestation schemes, such as [Bra+15; Eld+12; Koe+14b], they could leverage our Boot Attestation scheme to be applicable to a broader range of embedded devices.

The field of secure code updates specifically addresses the challenge of verifying the integrity firmware after it has been updated. Initial approaches employ software-based attestation techniques [Ses+06], and hence inherit their characteristics, mentioned above. Later on, the notion of Proofs of Secure Erasure (PoSE) was introduced to secure code updates [KL15; PT10]. PoSE-based approaches build on a challenge-response protocol that requires the prover to fill its entire memory with data, in turn overwriting any malicious code. Although such solutions can be applied to many devices, as they require a small amount of read-only memory, they assume that a network adversary only communicates with the verifier but not with the prover device, which is a strong limitation. Recent work focuses on scalable updates in large mesh networks [KK16]. In contrast to our work, it imposes the use of asymmetric cryptography, involving a heavy computational overhead and a large memory footprint. There are also platform-specific security extensions such as cryptoBSL [Tex15] and STM32 PCROP [STM16]. While they are focused on secure boot and IP protection, it would be interesting to evaluate their use in the context of remote attestation and recovery.

8.2. System Model and Goals

In this chapter, we specify our system model, discuss the adversaries' capabilities and describe the general procedure of remote attestation.

8.2.1. System Model

For our solution we consider a setting with two parties, a verifier $\mathcal{V}$ and a prover $\mathcal{P}$. $\mathcal{V}$ is interested in validating whether $\mathcal{P}$ is in a *known-good* software state, and for this purpose engages in a remote attestation protocol with $\mathcal{P}$.

According to Section 2.2, $\mathcal{P}$ is modeled as a commodity, low-cost embedded device as it may be found in personal gadgets, or smart home and smart factory appliances. In order to minimize manufacturing cost and power consumption, such devices tend to be single-purpose Microcontroller Units (MCUs) with often just the minimum memory and compute capabilities required to meet their intended application (cf. Figure 2.1). When programmed via low-level interfaces such as Joint Test Action Group (JTAG), many devices also allow to customize this early stage(s) of boot. We will revisit this property when implementing our Root of Trust (RoT) in Section 8.5.

Note that in the IoT context, the attestation verifier $\mathcal{V}$ is typically the owner of $\mathcal{P}$ (e.g., fitness trackers or USB thumb drives) or an operator who is responsible for managing $\mathcal{P}$ on behalf of the owner (e.g., smart factory or smart city).

Further, such devices usually exhibit very limited communication, foremost in low-energy scenarios, such as devices that use Bluetooth-Low-Energy for exchanging information. This allows for communication patterns to be arranged in advance. As a consequence, generic communication and key exchange protocols with many participants are neither required nor supported.

8.2.2. Threat Model

The adversary $\mathcal{A}$ controls the communication between $\mathcal{V}$ and $\mathcal{P}$ and can compromise the firmware of $\mathcal{P}$ at will. In more detail, $\mathcal{A}$ is granted full control over the communication channel (Dolev-Yao model) and thus can eavesdrop, inject, modify, or delete any messages between $\mathcal{V}$ and $\mathcal{P}$. $\mathcal{A}$ can also compromise the higher-level firmware on $\mathcal{P}$, i.e., the MCU application, whereupon $\mathcal{A}$ has full control over the execution and can read from and write to any memory.

However, we assume that $\mathcal{A}$ is unable to bypass hardware protection mechanisms, such as reading data from memory regions that are explicitly protected by hardware. We also exclude a comprehensive discussion of physical attacks as these require an in-depth analysis of the particular hardware design and are outside the scope of this work. Instead, we consider only a simple hardware adversary who may attempt to access documented interfaces such as JTAG, and replace or manipulate external System on Chip (SoC) components like external memory or radios (cf. Figure 2.1). We also assume that the verifier does not collaborate with $\mathcal{A}$, in particular, $\mathcal{V}$ will not disclose the attestation key to $\mathcal{A}$.

8.2.3. Remote Attestation Game

Remote Attestation is a security scheme where a verifier $\mathcal{V}$ wants to gain assurance that the firmware state of the prover $\mathcal{P}$ has not been subject to compromise by $\mathcal{A}$. Following the common load-time attestation model [PMP11], we define the firmware state of $\mathcal{P}$ as an ordered set of k binary measurements $M = (m_1, m_2, \ldots, m_k)$ that are taken as $\mathcal{P}$ loads its firmware for execution. Since the chain of measurements is started by the platform's RoT, it is assumed that any modification to the firmware state is reliably reflected in at least one measurement m_x.

To gain assurance on the actual state M' of $\mathcal{P}$, $\mathcal{V}$ and $\mathcal{P}$ engage in a challenge-response protocol. This culminates in the construction of an attestation report $r \leftarrow attest_{\mathrm{AK}}(c, M')$ at $\mathcal{P}$, where c is a random challenge and AK is an attestation key agreed between $\mathcal{P}$ and $\mathcal{V}$. $\mathcal{V}$ accepts $\mathcal{P}$ as trustworthy, i.e., not compromised, if the response r is valid under chosen values (c, AK) and an expected known-good state M (i.e., $M' = M$).

8.3. Boot Attestation

In this chapter, we introduce our Boot Attestation concept and protocol, extract hardware requirements and analyze its security with regard to Section 8.2.3.

8.3.1. Implicit Chain of Trust

Traditional attestation schemes collect measurements in a secure environment, such as a TPM or Trusted Execution Environment (TEE), which can be queried at a later time to produce an attestation report. They support complex software stacks comprising a large set of measurements and allow a multitude of verifiers to request subsets of these measurements, depending on privacy and validation requirements.

In contrast, our approach is to authenticate measurements m_x of the next firmware stage x immediately into an authenticated state M_x, before handing control to the next firmware stage. In this way, m_x is protected from manipulations by any subsequently loaded firmware application. The new state M_x is generated pseudo-randomly and the previous state M_{x-1} is discarded. This prevents an adversary from reconstructing prior or alternative measurement states. The final state M_k, seen by the application, comprises an authenticated representation of the complete measurement chain for reporting to $\mathcal{V}$:

$$M_x \leftarrow PRF_{\mathrm{AK}}(M_{x-1}, m_x). \tag{8.1}$$

As typically no secure hardware is available to protect AK in this usage, we generate pseudo-random sub-keys AK_x and again discard prior keys AK_{x-1} before initiating stage x:

$$\mathrm{AK}_x \leftarrow KDF_{\mathrm{AK}_{x-1}}(m_x), \text{ with } \mathrm{AK}_0 \leftarrow \mathrm{AK}. \tag{8.2}$$

Note that we can instantiate Pseudo-Random Function (PRF) and Key Derivation Function (KDF) using a single Keyed-Hash Message Authentication Code (HMAC). The measurement state M_x has become implicit in AK_x and does not have to be recorded separately.

The approach is limited in the sense that the boot flow at $\mathcal{P}$ dictates the accumulation of measurements in one or more implicit trust chains M. However, for the small, single-purpose IoT platforms we target here, there is typically no need to attest subsets of the firmware state as this is possible with TPM Platform Configuration Register (PCR). The next section expands this idea into a full remote attestation protocol.

8.3.2. Remote Attestation Protocol

Figure 8.1 provides an overview of a possible remote attestation protocol utilizing the implicit chain of trust and a symmetric shared attestation key AK. On the right-hand side, the prover $\mathcal{P}$ builds its chain of trust from the Root of Trust to a possible stage 1 (boot loader) and stage 2 (application). Once booted, the prover may be challenged by $\mathcal{V}$ to report its firmware state by demonstrating possession of the implicitly authenticated measurement state AK_2.

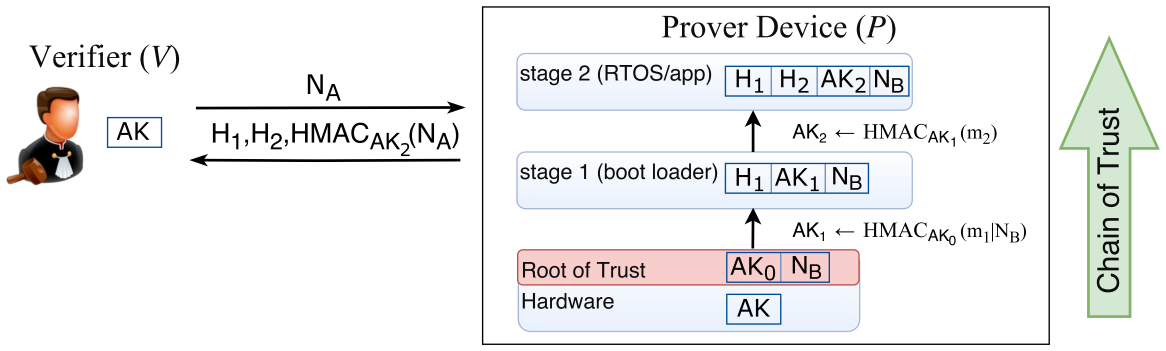


Figure 8.1.: Schematic overview of one possible instantiation of our Boot Attestation scheme as part of a remote attestation protocol.

The detailed protocol works as follows. The prover hardware starts execution at the platform RoT. This “stage 0” has exclusive access to the root attestation key $\mathrm{AK}_0 \leftarrow \mathrm{AK}$ and an optional boot nonce N_B. It derives AK_1 as $HMAC_{\mathrm{AK}_0}(N_B, m_1)$, with $m_1 := (start_1, size_1, H_1)$ defined as the binary measurement of the firmware stage 1. Before launching stage 1, the RoT must purge intermediate secrets from memory and lock AK against further access by firmware application. Execution then continues at stage 1 using the intermediate attestation key AK_1 and measurement log (H_1, N_B)[1].

The scheme continues through other boot stages $x \in \{1, \dots, k\}$ until the main application/runtime has launched in stage k. In each stage, a measurement m_{x+1} of the next firmware stage is taken and extended into the measurement state as $\mathrm{AK}_{x+1} \leftarrow HMACs_{\mathrm{AK}_x}(m_{x+1})$. The prior attestation key AK_x and intermediate values of the $HMAC()$ operation are purged from memory so that they cannot be accessed by stage $x+1$. Finally, the measurement log is extended to aid the later reconstruction and verification of the firmware state at $\mathcal{V}$.

Once $\mathcal{P}$ has launched the final stage k, it may accept challenges $c \leftarrow N_A$ by a remote verifier to attest its firmware state. For $\mathcal{P}$, this simply involves computing a proof of knowledge $r \leftarrow HMACs_{\mathrm{AK}_k}(N_A)$ and sending it to $\mathcal{V}$ together with the measurement log. Using this response, the verifier $\mathcal{V}$ can reconstruct the state $M' = (m'_1, \dots, m'_k)$ claimed by $\mathcal{P}$ and the associated AK_k. $\mathcal{V}$ can then validate and accept $\mathcal{P}$ if $M' = M$ and $r = HMACs_{\mathrm{AK}_k}(N_A)$.

Note that for the devices we target, k tends to be very low. Typical MCUs load only one or two stages of firmware, which helps keeping the validation effort at $\mathcal{V}$ manageable, even for large amounts of valid platforms (AK, M).

We emphasize that the protocol described here only considers the core attestation scheme. A complete solution should also consider authorizing $\mathcal{V}$ towards $\mathcal{P}$, protecting the confidentiality of the attestation report and linking the attestation to a session or otherwise exchanged data. As part of an authorized attestation challenge c', $\mathcal{V}$ may also include a command to update N_B and reboot $\mathcal{P}$ to refresh all AK_x. However, while the implications of managing N_B are discussed in Section 8.3.3, the detailed choices and goals are application-dependent and outside the scope of this work.

[1] We consider $(start_x, size_x)$ as well-known parameters here, since the individual m_x would typically encompass the complete firmware image at a particular stage.

8.3.3. Security Evaluation

In the following section, we analyze the security of Boot Attestation based on the adversary model and attestation game defined in Section 8.2.2 and 8.2.3. We will show that Boot Attestation is able to provide the same security as all load-time attestation approaches, such as TPMs-based attestation schemes [Tru11]. For this purpose, we consider the relevant attack surface in terms of *network*, *physical /side-channel* as well as *load-time* and *runtime* compromise attacks.

Network Attacks

The adversary $\mathcal{A}$ can eavesdrop, synthesize, manipulate, and drop any network data. However, the employed challenge-response protocol using the shared secret AK trivially mitigates these attacks. More specifically, any manipulation of assets exposed on the network, including H_1, H_2, N_A or the attestation response r, is detected by $\mathcal{V}$ when reconstructing AK_k and validating $r = HMACs_{\mathrm{AK}_k}(N_A)$. The attestation nonce N_A mitigates network replay attacks. $\mathcal{A}$ can cause a Denial of Service (DoS) by dropping messages, but $\mathcal{V}$ will still not accept $\mathcal{P}$.

Since AK is a device-specific secret, the intermediate keys AK_x and final response r are uniquely bound to each individual device. This allows Boot Attestation to function seamlessly with emerging swarm-attestation schemes, where the same nonce N_A is used to attest many devices at once [Amb+16; Aso+15; Car+17].

Physical and Side-Channel Attacks

$\mathcal{A}$ may attempt simple hardware attacks, using SoC-external interfaces to gather information on intermediate attestation keys AK_x or manipulate SoC-external memory and I/O. Boot Attestation assumes basic hardware mechanisms to protect debug ports and protect intermediate values in memory (cf. Section 8.4). Otherwise, the resilience against hardware attacks heavily depends on the SoCs implementation and is outside our scope.

$\mathcal{A}$ could also attempt software side-channel attacks, such as cache, data remanence, or timing attacks. However, as each stage i clears any data besides $\mathrm{N}, H_{i+1}, \mathrm{AK}_{i+1}$, there is no confidential data that a malware could extract from cache, RAM, or flash. Furthermore, the risk of timing side- channels is drastically reduced as root keys are only used initially by the RoT. Implementing a constant-time HMAC operation in the typically used non-paged, tightly coupled Static Random-Access Memory (SRAM) is straightforward.

Load-time Compromise

$\mathcal{A}$ may compromise the firmware stage a of $\mathcal{P}$ before it is loaded and hence, measured. In this case, $\mathcal{A}$ can access all intermediate measurements $(m_1, \ldots, m_k)$, the nonces (N_B, N_A), and any subsequent attestation keys $(\mathrm{AK}_a, \ldots, \mathrm{AK}_k)$. Note that compromising the RoT (i.e., the initial stage) is outside the capabilities of $\mathcal{A}$. This is a reasonable assumption due to RoT's hardware protection (cf. Section 8.4) and minuscule code complexity (cf. Table 8.1).

Compromising the intermediate measurement state and keys allows $\mathcal{A}$ to build alternative measurement states M'_{a+n} and associated attestation keys AK'_{a+n} for positive integers n. However, $\mathcal{A}$ cannot recover the attestation keys of prior stages $a - n$, as they have been wiped from memory prior to invoking stage a. In particular, $\mathcal{A}$ cannot access the root attestation key AK, which can only be accessed by the RoT. As a result, $\mathcal{A}$ can only construct attestation responses that *extend* on the measurement state M'_a and the associated attestation key AK_a. Moreover, load-time attestation assumes that the measurement chain is appropriately set up to record the compromise, so that (M_a', AK_a') already reflect the compromise and cannot be expanded to spoof a valid firmware state M_k.

In practice, successfully recording M_a' will typically require a persistent manipulation or explicit code loading action by the adversary. However, this is a well-known limitation of load-time attestation and also affects the TPMs and other load-time attestation schemes.

Following a firmware patch to return stage a into a well-known, trustworthy component, a new measurement and associated key chain is produced starting at stage a. $\mathcal{A}$ is unable to foresee this renewed key chain, as this would require access to at least AK_{a-1}.

Runtime Compromise

$\mathcal{A}$ may also compromise the firmware stage a at runtime, after it is measured, e.g., by exploiting a vulnerability that leads to arbitrary code execution. In this case, $\mathcal{A}$ would have access to the correct (unmodified) attestation key AK_a, could bypass the chain of trust, and thus win the attestation game. Note that this is a generic attack affecting all load-time attestation schemes, including the TPMs [Tru11]. Even platforms supporting a Dynamic Root of Trust Measurement (DRTM) cannot detect runtime attacks after the measurement was performed. However, Boot Attestation performs slightly worse in this case since $\mathcal{A}$ may additionally record AK_a and replay it later on to simulate alternative measurement states and win the attestation game, even after reboot. Nevertheless, Boot Attestation allows to recover the platform and to return into a trustworthy state, in the same way as with other load-time attestation schemes, by patching the vulnerable firmware stage a and performing a reboot. This leads to a refresh of attestation keys $\mathrm{AK}_a, \dots, \mathrm{AK}_k$ in a way that is unpredictable to $\mathcal{A}$, thus enabling $\mathcal{V}$ to attest the proper recovery of $\mathcal{P}$ and possibly re-provision application secrets.

To further mitigate the risk of a compromised AK_a, $\mathcal{V}$ may also manage an additional boot nonce N_B as introduced in Section 8.3. Depending on the particular usage and implementation of $\mathcal{P}$, N_B could be incremented to cause a refresh of the measurement chain without provisioning new firmware. For instance, MCUs in an automotive on-board network may regularly receive new boot nonces for use on next boot/validation cycle.

8.4. Hardware Requirements

In this chapter, we describe the hardware requirements of our Boot Attestation scheme in detail. We formulate these as results here and not as system assumptions in Section 8.2.1, since the exploration of alternative remote attestation schemes with minimal hardware requirements has been a major research challenge in recent years [Eld+12; Fra+14; Koe+14b; Noo+13]. In particular, remote attestation schemes proposed so far still require a secure co-processor or custom hardware security extensions in order to support the secure recording and signing of measurements. Alternative approaches using a software-only root of trust still require strong assumptions on the operating environment and implementation correctness, which has precluded them as a generic attestation solution for IoT [Cas+09; Li+15].

In particular, this requires (cf. Francillon et al. [Fra+14]):

- Exclusive Access to AK: the adversary would easily learn AK.

- No Leaks: the adversary would learn information about AK that could lead to an advantage in computing a valid r.
- Immutability: the adversary could change the code to move k to unprotected memory.
- Uninterruptibility: the adversary could move malware around during attestation, which helps escape detection.
- Controlled Invocation: the adversary could invoke Attest anywhere, which might cause it to be interruptible and/or skip sanity checks on input parameters.

Leveraging the implicit chain of trust, our Boot Attestation scheme avoids the requirement for a hardware-isolated attestation runtime. Specifically, we only require the following hardware security features:

RoT Integrity

The RoT is critical for initializing the chain of trust and protecting fundamental platform assets such as AK. Our scheme requires RoT integrity in the sense that it must be impossible for the adversary $\mathcal{A}$ to manipulate the RoT firmware, and that the RoT must be reliably and consistently executed at platform reset. In practice, this requires hardware access control on the RoT code and data region, but also hardware logic to consistently reset the SoCs's caches, DMA engines and other interrupt-capable devices in order to reliably execute RoT on power-cycle, soft-reset, deep sleep, and similar events.

While RoT integrity is well-understood in terms of supporting secure boot or firmware management, we know of no Commercial Off-The-Shelf (COTS) MCU which natively supports a RoT for attestation. To realize Boot Attestation on COTS MCUs we therefore require an extension of the RoT integrity requirement: The device owner must be able to customize or extend the initial boot phase to implement an attestation RoT, and then lock or otherwise protect it from any further manipulation. As we will show, many COTS MCUs actually offer this level of customization prior to enabling production use.

AK Protection

Boot attestation requires that the root attestation key AK is read-/write-locked ahead of execution of the firmware application. This typically requires the RoT to initialize some form of memory access control and then lock it down, so that it cannot be disabled by subsequent firmware stages. While such lock-able key storage is not a standard feature, we found that most COTS MCUs offer some kind of memory locking or hiding that can be used to meet this requirement (cf. Section 8.3.3).

AK Provisioning

Provisioning of a new attestation key AK_{new} involves replacement of its previous instance, conducted by the RoT. Hence, in order to support provisioning, AK must further be writable by the RoT exclusively. However, this procedure is preceded by the secure negotiation of AK_{new}. During this process the endorsement key EK is used to provide authorization and confidentiality of the new attestation key AK_{new}. Thus, during key provisioning the RoT must read EK and then lock it against read attempts by latter firmware stages, basically resembling requirement of AK protection.

State Protection

When calculating measurements m_x and attestation keys AK_x, the respective firmware stage must be able to operate in a secure memory that cannot be accessed by later firmware stages or other unauthorized platform components. This includes protecting intermediate values of the HMACs calculation as well as the stack. In practice, this requirement breaks down to operating in memory that is shielded against simple hardware attacks (cf. Section 8.2.2), such as the SoCs on-DIE SRAMs, and clearing sensitive intermediate values from memory before handing control to the respective next stage.

Debug Protection

Once programmed and provisioned, the device should reject unauthorized access via external interfaces such as UART consoles, JTAGs or SWD debug interfaces [CBW17]. Strictly speaking, this requirement is sufficiently addressed if the above integrity and confidentiality requirements are met. However, we list it here as separate requirement since debug access and re-programming protections are typically implemented and enabled separately from the above general implementation requirements.

Overall, we can see that Boot Attestation side-steps requirements for protecting the initial call into the secure environment and inhibiting interrupts during execution - including resets - which are not achievable with established hardware protection mechanisms and therefore also not feasible on commodity COTS MCUs [Eld+12; Fra+14].

8.5. Proof-of-Concept Implementation

We selected the Stellaris LM4F120H5QR to evaluate the implementation of our proposed scheme and to provide an overview of the associated costs. The implementation comprises extending the RoT for measuring the firmware application and deriving an attestation key, as well as initializing the hardware key protection for AK and EK. Note also that there is no intermediate boot loader stage on this device as the application image is directly executed by RoT (cf. Section 2.2.2). An overview of the implementation footprint is provided in Table 8.1. Note that a brief review of the datasheets of the device types evaluated for Physically Unclonable Function (PUF) behavior in this thesis (cf. Chapter 4.2) suggests that most of them also meet hardware requirements of Boot Attestation (cf. Section 8.4), including the ATMega328p, PIC16F1825, the PandaBoard as well as the Intel Galileo. Naturally, a full implementation and validation is required to ensure that the respective platform controls are accurately documented and sufficient in practice.

For the purpose of implementing a proof-of-concept, we leveraged FreeRTOS [Rea16] as a firmware stack, as it is freely available, pre-configured for the Stellaris and as it exhibits a small memory footprint.

Integrity Protected RoT

The Stellaris supports RoT code integrity by enabling execute-only protection to those flash blocks that store the boot loader. In particular, by setting register values of `FMPPEn` and `FMPREn` to ‘0’, read and write access to the boot loader section is disabled while keeping it executable.

Protection of AK & EK

Although the Stellaris provides memory protection for flash [Ash12], we decide not to use it for secure key storage. Despite that individual blocks of flash memory can be read-protected, it is yet possible to execute said blocks. This could enable an attacker $\mathcal{A}$ to extract bits of AK or EK. $\mathcal{A}$ can try to execute respective memory regions and infer information by interpreting resulting execution errors. Instead, we securely store AK and EK on the internal Electrically Erasable Programmable Read-only Memory (EEPROM) module. The Stellaris platform provides register `EEHIDE` that allows for individual 32 B EEPROM blocks to be hidden until subsequent reset.

PUF-based Storage of EK

It is also possible to securely store EK using a fraction of the on-chip SRAM as a PUF. Previous work supports the use of SRAM as PUFs for key storage [MTV09; SL12]. Indeed, the SRAM-based PUF instance of the Stellaris has already been characterized in [KSK15]. Using PUFs as a key storage significantly increases the level of protection, as PUF-based keys are existent only for a limited period [Arm+10]. Especially for long-term keys, such as EK, this is a desirable property, which is otherwise hard to achieve on low-cost devices. To evaluate this option, we implemented a Fuzzy Extractor construction based on [Bös+08]. On start-up of the device, a fraction of the SRAM start-up values are used as a (noisy) PUF measurement X. Using X and public Helper Data W that was created during a prior enrollment phase, the Fuzzy Extractor can reconstruct EK. For details on the interaction with SRAM-based PUFs, we refer to [Sch+14]. Assuming a conservative noise level of 15 % in the PUF measurements X, which is a common value used in literature [Bös+08], and applying a $(15, 1, 15)$ repetition code as part of the Fuzzy Extractor, we achieve an error probability of 10^{-9}.

Debug Protection

The boot loader is further protected from attempts to replace it by malicious code by disabling access to JTAG pins. For this purpose bits `DBG0`, `DBG1` and `NW`, part of register `BOOTCFG` are set to '0'. This leaves a subset of standard IEEE instructions intact (such as boundary scan operations), but blocks any access to the processor and peripherals.

8.6. Performance Evaluation

In the following section, we present evaluation result with focus on memory footprint and runtime. Numbers are given for the RoT and key protection logic. Values for the RoT are further separated with respect to RoT base logic (memory management, setup of data structures) and the HMAC implementation. Runtime results of the HMAC functionality are given for a memory range of 256 bit, i.e., a single HMAC data block, and a 32 kB region that reflects larger firmware measurements. For both memory footprint and runtime, we further provide numbers with respect to two different compile time optimizations. The detailed results are given in Table 8.1.

Memory

For memory consumption we consider static code segments (`.text`) and read-only data (`.rodata`) segments of the firmware image. Table 8.1 lists results for compile optimizations towards size (`-Os`) and runtime (`-O1`). Using the most memory-efficient setting, the scheme requires a total of about 3.1 kB on the Stellaris. This may seem large compared to the 700 B

	Size (Bytes)		**Runtime (ms)**	
Component	-0s	-01	-0s	-01
Base ROM	702	712	0.79	0.63
Root of Trust (RoT)				
Base Logic	336	340	< 0.01	< 0.01
*HMAC-SHA2 (*256 bit*)*	1828	1836	3.04	3.04
*HMAC-SHA2 (*32 kB*)*	1828	1836	312.26	312.26
AK Protection				
EEPROM	516	580	0.01	< 0.01
EK Protection				
EEPROM	516	580	0.01	< 0.01
SRAM PUF	1662	1980	46.44	46.42

Table 8.1.: Implementation overhead with respect to runtime in milliseconds (left) and memory overhead in Bytes (right) for the TI Stellaris (ARM), with compiler optimizations for size (**-0s**) and runtime (**-01**).

footprint of the base ROM image (i.e., excluding the firmware application), but is only 1.22 % of the total available flash.

Runtime

Additional runtime introduced by our scheme mainly results from HMAC operations in order to compute attestation measurements, with the key protection logic introducing only little overhead. The right-hand side of Table 8.1 lists runtime overheads of our implementation. As to be expected, the main overhead is caused by the HMAC function which depends on the concrete size of the next stage to be measured. We give 256 B and 32 kB as reference points to estimate the cost hashing the KDF output and a larger firmware, respectively. The Stellaris takes 312 ms for hashing 32 kB. In contrast, the key protection logic adds negligible runtime. It takes less than 0.02 ms in the worst-case. Lastly, the SRAM PUF is by far the slowest key storage solution for EK on the Stellaris, taking almost half a second. This is due to the costly error correction of the PUF measurements. As a reference, the unmodified base ROM, without our extension, takes on average 0.7 ms on the Stellaris and 6 ms. Note however, that key storage for EK can be implemented more efficiently on various other device types, leveraging their respective memory access control mechanisms.

8.7. Chapter Summary

In this chapter, we explored a novel lightweight remote attestation scheme for low-cost COTS MCUs. We showed that it is possible to narrow down hardware requirements of the targeted MCUs and even to enable the extension of already deployed devices. We demonstrated practicability and efficiency of implementing our scheme on two representative MCUs and proposed extensions for usage in real-world scenarios.

For future work, we will investigate the support of additional device types, to widen the scope of applicability. A second effort will be taken to develop protocol extensions, such as symmetric sealing of assets (i.e, sensor values, etc.), establishment of trusted channels or means to log provenance of such assets, especially if they are computed on flash-based media that employ our scheme.

Chapter 9

Conclusion

This chapter summarizes the main contributions of this thesis and concludes whether and to which extent the research question formulated at the beginning was answered and whether the research goal was accomplished.

The main research goal of this thesis was to establish trust in devices that belong to the class of low-cost commercial off-the-shelf microcontroller platforms, given that such devices usually lack the implementation of secure hardware. Hence these devices usually fail to provide decent security solutions albeit being used in privacy- and security-critical environments.

For the purpose of realizing the research goal, the concept of intrinsic memory-based Physically Unclonable Functions (PUFs) was used as a central means. Based on the concept of PUFs, the research question of this thesis was derived, which ought to answer whether variations in the physical behavior of standard memory components found on such Commercial Off-The-Shelf (COTS) embedded devices can be used as instances of PUFs and subsequently as a lightweight anchor of trust for security-related protocols.

In Chapter 4 we proved that PUF instances can indeed be found in standard memory components, namely Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory (DRAM), implemented on low-cost devices. In addition, in this chapter we showed that it is possible to successfully interface the PUF instances by software modifications only. Lastly, this chapter provided exhaustive evaluation results of the PUF characteristics that were proven to be of high quality, qualifying them for the subsequent implementation as hardware-building blocks of security applications. Hence, the existence of physical variations and the practicability to exploit them, i.e, to make them measurable by software, was successfully proven in this chapter.

Chapter 5 presented a first application of the previously found PUF instances. In particular, this chapter focused on the first phase of firmware execution, i.e., the boot phase. As was shown in this chapter, it is possible to interface intrinsic memory-based PUFs from within a modified commodity boot loader and to robustly derive a key that can be used to decrypt subsequent parts of the firmware. Hence, we showed that by relying on the concept of PUFs, it is possible to securely bootstrap firmware on low-cost devices.

Having considered the security of the boot phase of firmware execution, Chapter 6 dealt with the integrity of firmware in the subsequent stage of its actual execution. Therefore, a scheme was presented that achieves hardware-software binding and integrity protection at the same time by combining the concepts of PUFs and self-checksumming checking code regions. As was shown in this chapter, this method again is software-only, hence applicable to low-cost commodity devices and is at the same time efficient enough to impose only a negligible performance overhead.

Chapter 7 focused on the establishment of trust in the mutual authenticity of remote devices. For this purpose a lightweight mutual authentication protocol was presented. The protocol leverages the decay-based DRAM PUF presented earlier in this thesis and was specifically tailored to the highly constrained hardware capabilities of the targeted device class. In contrast to the majority of existing PUF-based authentication protocols, the proposed solution does not require error-correction, which has been shown to be too costly for implementation on low-cost devices.

Lastly, Chapter 8 covered the scenario of securely attesting the software state of a remote computing platform, which is a fundamental requirement in order to establish trust in interconnected devices. In particular, the chapter showed how on-board hardware primitives of low-cost devices can be exploited in order to realize a lightweight remote attestation protocol that further allows for safe recovery after potential device compromise. Along with other hardware primitives the solution leverages intrinsic memory-based PUFs to provide compatibility with a wide spectrum of IoT devices and legacy sensors.

We conclude that in this thesis, it was shown successfully that the concept of PUFs, with special focus on intrinsic memory-based PUF types, is an attractive and efficient solution to implement a hardware-based security anchor on otherwise unprotected low-cost devices and hence to allow for the implementation of lightweight security protocols and applications. The favorable properties of intrinsic memory-based PUFs in particular, namely their inherent existence in standard hardware components as well as their uniqueness and identifying properties, make them a natural candidate for establishing trust anchors on otherwise unprotected low-cost devices. As was demonstrated in this thesis, these PUF instances can be instantiated by software-only means, hence no hardware modifications are needed. In this way, legacy devices can be supported and even retro-fitting PUF-based security solutions to devices which have already been deployed is allowed. Thus, relying on intrinsic memory-based PUFs is the most attractive solution for establishing trust on such devices that fail to implement traditional security solutions.

Future Work

Although this thesis strives to be a comprehensive treatise on the subject of securing embedded devices via intrinsic memory-based PUFs, working on this topic led to further fields of research. These fields themselves comprise so many aspects that each of them individually could produce research results similar to the scope of this thesis at hand. Given the time limits of the process of the doctoral research process, these topics were not considered at a level of detail that would be appropriate. Nevertheless, the following section gives pointers to associated research fields that naturally follow the topics discussed in this thesis.

While the nature of the content of this thesis is primarily constructive, i.e., the majority of solutions presented strive for establishing trust and not attacking existing schemes, the logical next step is to analyze the resistance of the presented solutions against attacks. In particular, given the targeted devices types and the presented implementations of security protocols and applications, it would be illustrative to observe how effective different types of attacks can be. Given the often observed non-uniformity of PUF responses and high biases within these, an in-depth research of potential information leakage due to the applied Helper Data system (i.e., the Helper Data itself) would be imperative prior to productive deployment. Furthermore, the investigation of side-channel attacks, especially during the reconstruction phase of the fuzzy extraction of secrets, can reveal how the implementation of the PUF interface can be optimized towards the physical properties of the underlying hardware to be side-channel resistant. Lastly, although there are already works in the literature that deal with invasive attacks against particular PUF implementations, there is a lack of contributions that deal with highly integrated components, which are usually found on microcontroller and System on Chip (SoC)-based platforms. The memory components that are used in this thesis for instantiation of PUF instances are usually on-die and on modern SoC devices, they are organized in a Package-on-Package (PoP) fashion, i.e., they observe a stacked “horizontal” structure, which intuitively complicates known invasive attacks and might mitigate some of them entirely. Finally, a comprehensive analysis of attacks, the involved efforts and their success rate would allow for a detailed physical attacker model to be formulated that goes beyond the notion of zero security in the presence of such an adversary, that is used in many works [Hel+13]. Knowing lower bounds on the time- and resource-wise effort of different types of attacks would allow for designing security protocols in such a way that takes such bounds into consideration, whereas currently, respective protocols have to rely on rather vague assumptions [Ibr+16].

A second field of research that joins the matters discussed in this thesis is about the integration of intrinsic PUFs in classical security protocols. In particular, while PUFs have been proposed to be used as building blocks for basic security protocols, such as oblivious transfer [Rüh10; RD13], bit commitment schemes [Brz+11] and key exchange [DR12], these works usually treat PUFs as an abstract black box that omit peculiarities of different PUF types that indeed have an influence on the design of the constitutive protocols. Hence, it is desirable to have a thorough understanding of the practicability to use actual implementations of intrinsic PUFs as building blocks (i.e. as sources of entropy) for such basic security protocols and how their hardware related side-effects affect the security proofs and the overall design of respective protocols.

Bibliography

[Aic15] Barbara Aichinger. *DDR Memory Errors Caused by Row Hammer*. In: *High Performance Extreme Computing Conference*. 2015, pp. 1–5.

[Amb+16] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. *SANA: Secure and Scalable Aggregate Network Attestation*. In: *Conference on Computer & Communications Security*. 2016, pp. 731–742.

[Ana+13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. *Innovative Technology for CPU Based Attestation and Sealing*. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Vol. 13. 2013.

[AGS12] Otilia Anton, Brice Gelineau, and Jérémy Sauget. *Firmware and bootloader*. 2012.

[ARM17] ARM. *ARM TrustZone*. Last accessed on June 13th 2017. 2017. URL: `http://www.arm.com/products/processors/technologies/trustzone/index.php`.

[AHM14] Frederik Armknecht, Matthias Hamann, and Vasily Mikhalev. *Lightweight Authentication Protocols on Ultra-Constrained RFIDs-Myths and Facts*. In: *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. 2014, pp. 1–18.

[Arm+11] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, François-Xavier Standaert, and Christian Wachsmann. *A Formal Foundation for the Security Features of Physical Functions*. In: *Security & Privacy*. 2011, pp. 397–412.

[Arm+10] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Berk Sunar, and Pim Tuyls. *Memory Leakage-Resilient Encryption Based on Physically Unclonable Functions*. In: *Towards Hardware-intrinsic Security*. 2010, pp. 135–164.

[Arm+16] Frederik Armknecht, Daisuke Moriyama, Ahmad-Reza Sadeghi, and Moti Yung. *Towards a Unified Security Model for Physically Unclonable Functions*. In: *Cryptographers' Track at the RSA Conference*. 2016, pp. 271–287.

[Arm+13] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. *A Security Framework for the Analysis and Design of Software Attestation*. In: *Conference on Computer & Communications Security*. 2013, pp. 1–12.

[Ash12] Ashish Ahuja. *SPMA044A – Using Execute, Write, and Erase-Only Flash Protection on Stellaris Microcontrollers Using Code Composer Studio*. 2012. URL: `www.ti.com/lit/pdf/spma044`.

[Aso+15] N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. *Seda: Scalable Embedded Device Attestation*. In: *Conference on Computer & Communications Security*. 2015, pp. 964–975.

[Atm16] Atmel. *ATmega328P*. Last accessed on May 17th 2017. 2016. URL: `http://www.atmel.com/devices/atmega328p.aspx?tab=parameters`.

[Awe+16] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. *ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks*. In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 743–755.

[Bar16] Elaine B Barker. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. Tech. rep. National Institute of Standard and Technology, Aug. 2016. DOI: 10.6028/NIST.SP.800-175B.

[BK12] Elaine B Barker and John Michael Kelsey. *Recommendation for the Entropy Sources Used for Random Bit Generator*. US Department of Commerce, National Institute of Standards and Technology, 2012.

[Bar+10] John Barth, Don Plass, Erik Nelson, Charlie Hwang, Gregory Fredeman, Michael Sperling, Abraham Mathews, William Reohr, Kavita Nair, and Nianzheng Cao. *A 45nm SOI Embedded DRAM Macro for POWER7TM 32MB On-Chip L3 Cache*. In: *Solid-state Circuits Conference Digest of Technical Papers, IEEE International*. 2010, pp. 342–343.

[BM13] Mudit Bhargava and Ken Mai. *A High Reliability PUF Using Hot Carrier Injection Based Response Reinforcement*. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 2013, pp. 90–106.

[BM16] Sarani Bhattacharya and Debdeep Mukhopadhyay. *Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis*. In: *International Conference on Cryptographic Hardware and Embedded Systems*. 2016, pp. 602–624.

[BR07] Leonid Bolotnyy and Gabriel Robins. *Physically Unclonable Function-Based Security and Privacy in RFID Systems*. In: *Fifth Annual IEEE International Conference on Pervasive Computing and Communications.* 2007, pp. 211–220.

[BEK14] Carsten Bormann, Mehmet Ersue, and A. Keranen. *Terminology for Constrained-Node Networks*. Tech. rep. 2014.

[Bös+08] Christoph Bösch, Jorge Guajardo, Ahmad-Reza Sadeghi, Jamshid Shokrollahi, and Pim Tuyls. *Efficient Helper Data Key Extractor on FPGAs*. In: *Cryptographic Hardware and Embedded Systems*. 2008, pp. 181–197.

[BP92] Antoon Bosselaers and Bart Preneel. *SKID*. In: *Integrity Primitives for Secure Information Systems: Final Report of RACE Integrity Primitives Evaluation RIPE-RACE* 1040 (1992), pp. 169–178.

[Boy04] Xavier Boyen. *Reusable Cryptographic Fuzzy Extractors*. In: *Proceedings of the 11th Acm Conference on Computer and Communications Security*. 2004, pp. 82–91.

[Bra+15] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. *Tytan: Tiny Trust Anchor for Tiny Devices*. In: *Design Automation Conference, 52nd Acm/edac/ieee*. 2015, pp. 1–6.

[BCD06] Julien Bringer, Hervé Chabanne, and Emmanuelle Dottax. *HB^{++}: a Lightweight Authentication Protocol Secure against Some Attacks*. In: *Second International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing,* 2006, pp. 28–33.

[Bro09] Daniel R L Brown. *Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography*. In: *Released Standard Version* 1 (2009).

[Brz+11] Christina Brzuska, Marc Fischlin, Heike Schröder, and Stefan Katzenbeisser. *Physically Uncloneable Functions in the Universal Composition Framework.* In: *Crypto*. Vol. 6841. 2011, pp. 51–70.

[b313] bunnie and xobs at 30C3. *SD Card Hacking*. Last accessed 19th April 2017. 2013. URL: `https://bunniefoo.com/bunnie/sdcard-30c3-pub.pdf`.

[BAN89] Michael Burrows, Martin Abadi, and Roger M Needham. *A Logic of Authentication*. In: *Proceedings of the Royal Society of London a: Mathematical, Physical and Engineering Sciences*. Vol. 426. 1871. 1989, pp. 233–271.

[Car+17] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. *Lightweight Swarm Attestation: A Tale of Two LISA-s*. In: *Proceedings of the 2017 Acm on Asia Conference on Computer and Communications Security*. 2017, pp. 86–100.

[Cas+09] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. *On the Difficulty of Software-Based Attestation of Embedded Devices*. In: *Conference on Computer & Communications Security*. 2009, pp. 400–409.

[CBW17] Wen Chen, Jayanta Bhadra, and Li-C Wang. *SoC Security and Debug*. In: *Fundamentals of Ip and Soc Security*. 2017, pp. 29–48.

[Chi07] Hung-Yu Chien. *SASI: A New Ultralightweight RFID Authentication Protocol Providing Strong Authentication and Strong Integrity*. In: *IEEE Transactions on Dependable and Secure Computing* 4.4 (2007), pp. 337–340.

[CLB11] Mathias Claes, Vincent van der Leest, and An Braeken. *Comparison of SRAM and FF PUF in 65nm Technology*. In: *Nordic Conference on Secure It Systems*. 2011, pp. 47–64.

[Cor09] Thomas H Cormen. *Introduction to Algorithms*. MIT press, 2009.

[CD16] Victor Costan and Srinivas Devadas. *Intel SGX Explained.* In: *IACR Cryptology ePrint Archive* (2016), p. 86.

[Cos+14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. *A Large-Scale Analysis of the Security of Embedded Firmwares.* In: *Usenix Security Symposium*. 2014, pp. 95–110.

[CT12] Thomas M Cover and Joy A Thomas. *Elements of Information Theory*. John Wiley & Sons, 2012.

[Def15] Hacking DefCon. *Hacking DefCon 23' s IoT Village Samsung fridge*. Last accessed on July 19th 2017. 2015.

[DR12] Marten van Dijk and Ulrich Rührmair. *Physical Unclonable Functions in Cryptographic Protocols: Security Proofs and Impossibility Results.* In: *IACR Cryptology ePrint Archive* (2012), p. 228.

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. *Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data*. In: *Advances in Cryptology-eurocrypt*. 2004, pp. 523–540.

[DY83] Danny Dolev and Andrew C Yao. *On the Security of Public Key Protocols*. In: *Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[Edi16] Edition, NXP LPC17xx. *Using the FreeRTOS™Real Time Kernel*. Last accessed on October 9th 2017. 2016. URL: `https://www.nxp.com/wcm_documents/techzones/microcontrollers-techzone/LPCLibrary-Books/Books-pdf/using.freertos.lpc17xx.summary.pdf`.

[Eir+12] Susana Eiroa, J Castro, Macarena Cristina Martínez-Rodríguez, Erica Tena, Piedad Brox, and Iluminada Baturone. *Reducing Bit Flipping Problems in SRAM Physical Unclonable Functions for Chip Identification*. In: *19th IEEE International Conference on Electronics, Circuits and Systems*. 2012, pp. 392–395.

[EK07] Thomas Eisenbarth and Sandeep Kumar. *A Survey of Lightweight-Cryptography Implementations*. In: *IEEE Design & Test of Computers* 24.6 (2007).

[Eld+12] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. *SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust*. In: *19th Annual Network and Distributed System Security Symposium*. Vol. 12. 2012, pp. 1–15.

[Eng02] DENX Software Engineering. *Das U-Boot–the Universal Boot Loader*. Last accessed on October 10th 2017. 2002. URL: `http://www.denx.de/wiki/U-Boot`.

[Fos+15] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. *Fast and VulnerableV: a Story of Telematic Failures*. In: *Usenix Workshop on Offensive Technologies*. 2015.

[Fra+14] Aurélien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. *A Minimalist Approach to Remote Attestation*. In: *Design, Automation & Test in Europe*. 2014, 244:1–244:6.

[Fre+15] Gregory Fredeman, Donald Plass, Abraham Mathews, Kenneth Reyer, Thomas Knips, Thomas Miller, Elizabeth Gerhard, Dinesh Kannambadi, Chris Paone, Dongho Lee, et al. *17.4 A 14nm 1.1 Mb Embedded DRAM Macro with 1ns Access*. In: *Solid-state Circuits Conference, IEEE International*. 2015, pp. 1–3.

[Gan+17] Fatemeh Ganji, Shahin Tajik, Fabian Fäßler, and Jean-Pierre Seifert. *Having No Mathematical Model May not Secure PUFs*. In: *Journal of Cryptographic Engineering* (2017), pp. 1–16.

[GTS16] Fatemeh Ganji, Shahin Tajik, and Jean-Pierre Seifert. *PAC Learning of Arbiter PUFs*. In: *Journal of Cryptographic Engineering* 6.3 (2016), pp. 249–258.

[GK14] Achiranshu Garg and Tony T. Kim. *Design of SRAM PUF with Improved Uniformity and Reliability Utilizing Device Aging Effect*. In: *2014 IEEE International Symposium on Circuits and Systems*. 2014, pp. 1941–1944.

[Gas+02] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. *Silicon Physical Random Functions*. In: *Proceedings of the 9th Acm Conference on Computer and Communications Security*. 2002, pp. 148–160.

[Gas+07] Blaise Gassend, Marten van Dijk, Dwaine Clarke, and Srinivas Devadas. *Controlled Physical Random Functions*. In: *Security with Noisy Data*. 2007, pp. 235–253.

[Ger+14] Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. *Internet of Vehicles: From Intelligent Grid to Autonomous Cars and Vehicular Clouds*. In: *IEEE World Forum on Internet of Things*. 2014, pp. 241–246.

[GRS08] Henri Gilbert, Matthew JB Robshaw, and Yannick Seurin. *Good Variants of HB+ are Hard to Find*. In: *International Conference on Financial Cryptography and Data Security*. 2008, pp. 156–170.

[GRS05] Henri Gilbert, Matthew Robshaw, and Herve Sibert. *Active Attack Against HB/sup+: a Provably Secure Lightweight Authentication Protocol*. In: *Electronics Letters* 41.21 (2005), pp. 1169–1170.

[Goo09] Travis Goodspeed. *Extracting Keys from Second Generation Zigbee Chips*. In: *Black Hat USA* 9 (2009).

[GMS09] Michael A Gora, Abhranil Maiti, and Patrick Schaumont. *A Flexible Design Flow for Software IP Binding in Commodity FPGA*. In: *IEE International Symposium on Industrial Embedded Systems*. 2009, pp. 211–218.

[Gre15] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway–With Me in it*. In: *Wired (Online), 21 July* (2015). Last accessed on July 19th 2017. URL: `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`.

[Gua+07] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. *FPGA Intrinsic PUFs and Their Use for IP Protection*. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 2007, pp. 63–80.

[HSS98] Takeshi Hamamoto, Soichi Sugiura, and Shizuo Sawada. *On The Retention Time Distribution of Dynamic Random Access Memory (DRAM)*. In: *IEEE Transactions on Electron devices* 45.6 (1998), pp. 1300–1309.

[Has+15] Maryam S Hashemian, Bhanu Singh, Francis Wolff, Daniel Weyer, Steve Clay, and Christos Papachristou. *A Robust Authentication Methodology Using Physically Unclonable Functions in DRAM Arrays*. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. 2015, pp. 647–652.

[Hel+13] Clemens Helfmeier, Christian Boit, Dmitry Nedospasov, and Jean-Pierre Seifert. *Cloning Physically Unclonable Functions*. In: *Hardware-oriented Security and Trust*. 2013, pp. 1–6.

[HM04] Dirk Henrici and Paul Muller. *Hash-Based Enhancement of Location Privacy for Radio-Frequency Identification Devices Using Varying Identifiers*. In: *IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second*. 2004, pp. 149–153.

[Her+14] Grant Hernandez, Orlando Arias, Daniel Buentello, and Yier Jin. *Smart Nest Thermostat: A Smart Spy in Your Home*. In: *Black Hat USA* (2014).

[HV14] Anthony van Herrewege and Ingrid Verbauwhede. *Software Only, Extremely Compact, Keccak-based Secure PRNG on ARM Cortex-M*. In: *ACM Proceedings of the 51st Annual Design Automation Conference*. 2014.

[Her+09] Amir Herzberg, Haya Shulman, Amitabh Saxena, and Bruno Crispo. *Towards a Theory of White-Box Security*. In: *Emerging Challenges for Security, Privacy and Trust*. 2009.

[HBF09] Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. *Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers*. In: *IEEE Transactions on Computers* 58.9 (2009), pp. 1198–1210.

[HF14] Daniel E Holcomb and Kevin Fu. *Bitline PUF: Building Native Challenge-Response PUF Capability into any SRAM*. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 2014, pp. 510–526.

[Hol+12] Daniel E Holcomb, Amir Rahmati, Mastooreh Salajegheh, Wayne P Burleson, and Kevin Fu. *DRV-Fingerprinting: Using Data Retention Voltage of SRAM Cells for Chip Identification*. In: *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. 2012, pp. 165–179.

[H+07] Daniel E Holcomb, Wayne P Burleson, Kevin Fu, et al. *Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags*. In: *Proceedings of the Conference on RFID Security*. Vol. 7. 2007, p. 2.

[Hon+06] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, et al. *HIGHT: A New Block Cipher Suitable for Low-Resource Device*. In: *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*. Vol. 4249. 2006, pp. 46–59.

[HB01] Nicholas J Hopper and Manuel Blum. *Secure Human Identification Protocols*. In: *International Conference on the Theory and Application of Cryptology and Information Security*. 2001, pp. 52–66.

[Hor+02] Bill Horne, Lesley Matheson, Casey Sheehan, and Rober Tarjan. *Dynamic Self-Checking Techniques for Improved Tamper Resistance*. In: *ACM Workshop on Security and Privacy in Digital Rights Management*. 2002.

[HMV12] Gabriel Hospodar, Roel Maes, and Ingrid Verbauwhede. *Machine Learning Attacks on 65nm Arbiter PUFs: Accurate Modeling Poses Strict Bounds on Usability*. In: *IEEE International Workshop on Information Forensics and Security*. 2012, pp. 37–42.

[HLN17] Galen Hunt, George Letey, and Ed Nightingale. *The Seven Properties of Highly Secure Devices*. Tech. rep. 2017.

[Ibr+16] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. *DARPA: Device Attestation Resilient to Physical Attacks*. In: *Proceedings of the 9th Acm Conference on Security & Privacy in Wireless and Mobile Networks*. 2016, pp. 171–182.

[Ign+06] Tanya Ignatenko, Geert-Jan Schrijen, Boris Skoric, Pim Tuyls, and Frans Willems. *Estimating the Secrecy-rate of Physical Unclonable Functions with the Context-tree Weighting Method*. In: *IEEE International Symposium on Information Theory*. 2006, pp. 499–503.

[Int16a] Intel. *Intel® Galileo Gen 2 Board*. Last accessed on September 21st 2017. 2016. URL: `https://ark.intel.com/products/83137/Intel-Galileo-Gen-2-Board`.

[Int16b] Intel. *Intel® Software Guard Extensions (Intel® SGX)*. Last accessed on August 07th 2017. 2016. URL: `https://software.intel.com/en-us/sgx`.

[Int17] Intrinsid ID. *Intrinsic ID – IoT Security*. Last accessed on September 13th 2017. 2017. URL: `https://www.intrinsic-id.com/`.

[Iye+05] Subramanian S Iyer, JE Barth, Paul C Parries, James P Norum, James P Rice, Lyndon R Logan, and Dennis Hoyniak. *Embedded DRAM: Technology Platform for the Blue Gene/L Chip*. In: *IBM Journal of Research and Development* 49.2.3 (2005), pp. 333–350.

[Jaz14] Nasser Jazdi. *Cyber Physical Systems in the Context of Industry 4.0*. In: *IEEE International Conference on Automation, Quality and Testing, Robotics*. 2014, pp. 1–4.

[Jue+04] Ari Juels et al. *Minimalist Cryptography for Low-Cost RFID Tags.* In: *Proceedings of the 4th International Conference on Security in Communication Networks*. Vol. 3352. 2004, pp. 149–164.

[JP03] Ari Juels and Ravikanth Pappu. *Squealing Euros: Privacy Protection in Rfid-enabled Banknotes*. In: *Computer Aided Verification*. 2003, pp. 103–121.

[J+05] Ari Juels, Stephen A Weis, et al. *Authenticating Pervasive Devices with Human Protocols*. In: *Proceedings of the 25th Annual International Conference on Advances in Cryptology*. Vol. 3621. 2005, pp. 293–308.

[Kal+10] Ron Kalla, Balaram Sinharoy, William J Starke, and Michael Floyd. *Power7: IBM's Next-Generation Server Processor*. In: *IEEE micro* 30.2 (2010).

[KL15] Ghassan O Karame and Wenting Li. *Secure Erasure and Code Update in Legacy Sensors*. In: *Trust and Trustworthy Computing*. 2015, pp. 283–299.

[Kat+12] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. *PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon*. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 2012, pp. 283–301.

[Kat+11] Stefan Katzenbeisser, Ünal Koçabas, Vincent Van Der Leest, Ahmad-Reza Sadeghi, Geert-Jan Schrijen, Heike Schröder, and Christian Wachsmann. *Recyclable Pufs: Logically Reconfigurable PUFs*. In: *International Conference on Cryptographic Hardware and Embedded Systems* (2011), pp. 374–389.

[Kee08] Brent Keeth. *DRAM Circuit Design: Fundamental and High-Speed Topics*. Vol. 13. John Wiley & Sons, 2008.

[Kel+14] Christoph Keller, Frank Gurkaynak, Hubert Kaeslin, and Norbert Felber. *Dynamic Memory-Based Physically Unclonable Function for the Generation of unique Identifiers and True Random Numbers*. In: *Circuits and Systems, IEEE International Symposium On*. 2014, pp. 2740–2743.

[Kim+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors*. In: *Acm Sigarch Computer Architecture News*. Vol. 42. 3. 2014, pp. 361–372.

[Koç+11] Ünal Koçabas, Ahmad Reza Sadeghi, Christian Wachsmann, and Steffen Schulz. *Poster: Practical Embedded Remote Attestation Using Physically Unclonable Functions*. In: *Proceedings of the 18th Acm Conference on Computer and Communications Security*. 2011, pp. 797–800.

[Koe+14a] Patrick Koeberl, Jiangtao Li, Anand Rajan, and Wei Wu. *Entropy Loss in PUF-Based Key Generation Schemes: The Repetition Code Pitfall*. In: *IEEE International Symposium on Hardware-oriented Security and Trust.* 2014, pp. 44–49.

[Koe+14b] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. *TrustLite: A Security Architecture for Tiny Embedded Devices*. In: *European Conference on Computer Systems*. 2014, 10:1–10:14.

[KK16] Florian Kohnhäuser and Stefan Katzenbeisser. *Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices*. In: *European Symposium on Research in Computer Security*. 2016, pp. 320–338.

[KSK15] Florian Kohnhäuser, André Schaller, and Stefan Katzenbeisser. *PUF-Based Software Protection for Low-End Embedded Devices*. In: *Trust and Trustworthy Computing*. 2015, pp. 3–21.

[kok17] kokke. *kokke/tiny-AES-c: Small portable AES128/192/256 in C*. Last accessed on October 16th 2017. 2017. URL: `https://github.com/kokke/tiny-AES-c`.

[Kon+14] Joonho Kong, Farinaz Koushanfar, Praveen K Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. *PUFatt: Embedded Platform Attestation Based on Novel Processor-Based PUFs*. In: *Proceedings of the 51st Annual Design Automation Conference*. 2014, pp. 1–6.

[Kov+12] Xeno Kovah, Corey Kallenberg, Chris Weathers, Alexander Herzog, Matthew Albin, and John Butterworth. *New Results for Timing-Based Attestation*. In: *Security & Privacy*. 2012, pp. 239–253.

[Kum+08] Sandeep S Kumar, Jorge Guajardo, Roel Maes, Geert-Jan Schrijen, and Pim Tuyls. *The Butterfly PUF Protecting IP on Every FPGA*. In: *IEEE International Workshop on Hardware-oriented Security and Trust*. 2008, pp. 67–70.

[Lan16] Mark Lanteigne. *How Rowhammer Could be Used to Exploit Weaknesses in Computer Hardware*. Last accessed on September 25th 2017. 2016. URL: `http://www.thirdio.com/rowhammer.pdf`.

[Lar+14] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. *SoK: Automated Software Diversity*. In: *IEEE Symposium on Security and Privacy*. 2014.

[LA14] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In: *IEEE Symposium on Code Generation and Optimization*. 2014.

[Laz96] Felix Lazebnik. *On Systems of Linear Diophantine Equations*. In: *Mathematics Magazine*. 1996.

[LS16] Edward Ashford Lee and Sanjit A Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. MIT Press, 2016.

[Lee+04] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. *A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Applications*. In: *Symposium on Vlsi Circuits*. 2004, pp. 176–179.

[LMA16] Robert P Lee, Konstantinos Markantonakis, and Raja Naeem Akram. *Binding Hardware and Software to Prevent Firmware Modification and Device Counterfeiting*. In: *Proceedings of the 2nd Acm International Workshop on Cyber-physical System Security*. 2016, pp. 70–81.

[Lee+05] Su Mi Lee, Young Ju Hwang, Dong Hoon Lee, and Jong In Lim. *Efficient Authentication for Low-Cost RFID Systems*. In: *International Conference on Computational Science and Its Applications*. 2005, pp. 619–627.

[Lee+12] Vincent van der Leest, Erik van der Sluis, Geert-Jan Schrijen, Pim Tuyls, and Helena Handschuh. *Efficient Implementation of True Random Number Generator Based on SRAM PUFs*. In: *Cryptography and Security: From Theory to Applications*. Springer, 2012, pp. 300–318.

[LT13] Vincent van der Leest and Pim Tuyls. *Anti-Counterfeiting with Hardware Intrinsic Security*. In: *Design, Automation & Test in Europe Conference & Exhibition*. 2013, pp. 1137–1142.

[LCK16] Paulo Leitão, Armando Walter Colombo, and Stamatis Karnouskos. *Industrial Automation Based on Cyber-Physical Systems Technologies: Prototype Implementations and Challenges*. In: *Computers in Industry* 81 (2016), pp. 11–25.

[LTL16] Shancang Li, Theo Tryfonas, and Honglei Li. *The Internet of Things: A Security Point of View*. In: *Internet Research* 26.2 (2016), pp. 337–359.

[Li+15] Yanlin Li, Yueqiang Cheng, Virgil Gligor, and Adrian Perrig. *Establishing Software-only Root of Trust on Embedded Systems: Facts and Fiction*. In: *International Workshop on Security Protocols*. 2015, pp. 50–68. ISBN: 978-3-319-26096-9. DOI: `10.1007/978-3-319-26096-9_7`. URL: `http://www.netsec.ethz.ch/publications/papers/SWORT-SPW-2015.pdf`.

[LMP11] Yanlin Li, Jonathan M McCune, and Adrian Perrig. *VIPER: verifying the integrity of PERipherals' firmware*. In: *Conference on Computer & Communications Security*. 2011, pp. 3–16.

[Lin+10] Lang Lin, Dan Holcomb, Dilip Kumar Krishnappa, Prasad Shabadi, and Wayne Burleson. *Low-Power Sub-Threshold Design of Secure Physical Unclonable Functions*. In: *Proceedings of the 16th Acm/ieee International Symposium on Low Power Electronics and Design*. 2010, pp. 43–48.

[Liu+14] Wenchao Liu, Zhenhua Zhang, Miaoxin Li, and Zhenglin Liu. *A Trustworthy Key Generation Prototype Based on DDR3 PUF for Wireless Sensor Networks*. In: *Sensors* 14.7 (2014), pp. 11542–11556.

[LGK10] Zhe Liu, Johann Großschädl, and Ilya Kizhvatov. *Efficient and Side-Channel Resistant RSA Implementation for 8-Bit AVR Microcontrollers*. In: *Workshop on the Security of the Internet of Things*. Vol. 10. 2010.

[LLV07] LLVM. *Clang: A C Language Family Frontend for LLVM*. Last accessed on September 25th 2017. 2007. URL: `http://clang.llvm.org/`.

[Mae10] Maes, Roal and Verbauwhede, Ingrid. *Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions*. In: *Towards Hardware-intrinsic Security*. 2010, pp. 3–37.

[ML14] Roel Maes and Vincent van der Leest. *Countering the Effects of Silicon Aging on SRAM PUFs*. In: *IEEE International Symposium on Hardware-oriented Security and Trust*. 2014, pp. 148–153.

[Mae+12] Roel Maes, Vladimir Rozic, Ingrid Verbauwhede, Patrick Koeberl, Erik Van der Sluis, and Vincent van der Leest. *Experimental Evaluation of Physically Unclonable Functions in 65 nm CMOS*. In: *Proceedings of the Esscirc*. 2012, pp. 486–489.

[MTV08] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. *Intrinsic PUFs from Flip-Flops on Reconfigurable Devices*. In: *3rd Benelux Workshop on Information and System Security*. Vol. 17. 2008, p. 2008.

[MTV09] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. *Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs*. In: *Cryptographic Hardware and Embedded Systems*. 2009, pp. 332–347.

[MKP08] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. *Lightweight Secure PUFs*. In: *IEEE/acm International Conference on Computer-aided Design*. 2008, pp. 670–673.

[Mar17] Markku-Juhani O. Saarinen. *mjosaarinen/tiny_sha3: Very small, readable implementation of the SHA3 hash function*. Last accessed on October 16th 2017. 2017. URL: `https://github.com/mjosaarinen/tiny%5C_sha3`.

[MVV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 1996.

[Mic15] Microchip. *PIC16F1825*. Last accessed on May 17th 2017. 2015. URL: `http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en546902`.

[MP07] Jorge Munilla and Alberto Peinado. *HB-MP: A Further Step in the HB-Family of Lightweight Authentication Protocols*. In: *Computer Networks* 51.9 (2007), pp. 2262–2267.

[NC09] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[NW97] Roger M Needham and David J Wheeler. *Tea Extensions*. In: *Report, Cambridge University, Cambridge, UK* (1997).

[NT94] B Clifford Neuman and Theodore Ts'o. *Kerberos: An Authentication Service for Computer Networks*. In: *IEEE Communications magazine* 32.9 (1994), pp. 33–38.

[NJ16] Arsalan Mohsen Nia and Niraj K Jha. *A Comprehensive Study of Security of Internet-of-Things*. In: *IEEE Transactions on Emerging Topics in Computing* (2016).

[NL14] Karsten Nohl and Jakob Lell. *BadUSB-On Accessories That Turn Evil*. In: *Black Hat USA* (2014).

[Noo+13] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. *Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base*. In: *Usenix Security*. 2013, pp. 479–494.

[ÖHS08] Erdinç Öztürk, Ghaith Hammouri, and Berk Sunar. *Towards Robust Low Cost Authentication for Pervasive Devices*. In: *Sixth Annual IEEE International Conference on Pervasive Computing and Communications.* 2008, pp. 170–178.

[PMP11] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011. URL: `https://www.microsoft.com/en-us/research/publication/bootstrapping-trust-in-modern-computers/`.

[PTT17] Chris Pavlina, Jacob Torrey, and Kyle Temkin. *Characterizing EEPROM for Usage as a Ubiquitous PUF Source*. In: *IEEE International Symposium on Hardware Oriented Security and Trust.* 2017, pp. 168–168.

[Pei+14] C Pei, G Wang, M Aquilino, N Arnold, B Chandra, W Chang, X Chen, W Davies, K Hawkins, D Jaeger, et al. *0.026 µm 2 High Performance Embedded DRAM in 22nm Technology for Server and SOC Applications*. In: *IEEE International Electron Devices Meeting*. 2014, pp. 19–4.

[Per+06a] Pedro Peris-Lopez, Julio Cesar Hernandez-Castro, Juan M Estevez-Tapiador, and Arturo Ribagorda. *EMAP: An Efficient Mutual-Authentication Protocol for Low-Cost RFID Tags*. In: *Otm Confederated International Conferences "on the Move to Meaningful Internet Systems"*. 2006, pp. 352–361.

[Per+06b] Pedro Peris-Lopez, Julio Cesar Hernandez-Castro, Juan M Estévez-Tapiador, and Arturo Ribagorda. *LMAP: A Real Lightweight Mutual Authentication Protocol for Low-Cost RFID Tags*. In: *Proceedings of the 2nd Workshop on RFID Security*. 2006, p. 06.

[PT10] Daniele Perito and Gene Tsudik. *Secure Code Update for Embedded Devices via Proofs of Secure Erasure*. In: *Proceedings of the 15th European Conference on Research in Computer Security*. Vol. 6345. 2010, pp. 643–662.

[Pie00] Henna Pietiläinen. *Elliptic Curve Cryptography on Smart Cards*. In: *M. Sc., Helsinki Univ. of Technology* (2000).

[PLP06] Krzysztof Piotrowski, Peter Langendoerfer, and Steffen Peter. *How Public Key Cryptography Influences Wireless Sensor Node Lifetime*. In: *Proceedings of the Fourth Acm Workshop on Security of Ad Hoc and Sensor Networks*. 2006, pp. 169–176.

[Pra+11] Pravin Prabhu, Ameen Akel, Laura M Grupp, S Yu Wing-Kei, G Edward Suh, Edwin Kan, and Steven Swanson. *Extracting Device Fingerprints from Flash Memory by Exploiting Physical variations*. In: *International Conference on Trust and Trustworthy Computing*. 2011, pp. 188–201.

[Rah+15] Amir Rahmati, Matthew Hicks, Daniel E Holcomb, and Kevin Fu. *Probable Cause: The Deanonymizing Effects of Approximate DRAM*. In: *Acm Sigarch Computer Architecture News*. Vol. 43. 3. 2015, pp. 604–615.

[Rah+12] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P Burleson, and Kevin Fu. *TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks*. In: *Proceedings of the 21st Usenix Conference on Security Symposium*. 2012, pp. 36–36.

[Raz+16] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. *Flip Feng Shui: Hammering a Needle in the Software Sstack*. In: *Proceedings of the 25th Usenix Security Symposium*. 2016.

[Rea16] Real Time Engineers Ltd. *FreeRTOS Website*. Last accessed on December 9th 2015. 2016. URL: `http://www.freertos.org/`.

[Rén+61] Alfréd Rényi et al. *On Measures of Entropy and Information*. In: *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1. 1961, pp. 547–561.

[Roa+15] Austin H Roach, Matthew J Gadlage, Adam R Duncan, James D Ingalls, and Matthew J Kay. *Interrupted Program and Erase Operations for Characterizing Radiation Effects in Commercial Nand Flash Memories*. In: *IEEE Transactions on Nuclear Science* 62.6 (2015), pp. 2390–2397.

[Roe12] Maes Roel. *Physically Unclonable Functions: Constructions, Properties and Applications*. PhD thesis. Ph. D. thesis, Dissertation, University of KU Leuven, 2012.

[Ros+13a] Sami Rosenblatt, Srivatsan Chellappa, Alberto Cestero, Norman Robson, Toshiaki Kirihata, and Subramanian S Iyer. *A Self-Authenticating Chip Architecture Using an Intrinsic Fingerprint of Embedded DRAM*. In: *IEEE Journal of Solid-State Circuits* 48.11 (2013), pp. 2934–2943.

[Ros+13b] Sami Rosenblatt, Daniel Fainstein, Alberto Cestero, John Safran, Norman Robson, Toshiaki Kirihata, and Subramanian S Iyer. *Field Tolerant Dynamic Intrinsic Chip ID Using 32 nm high-K/metal gate SOI Embedded DRAM*. In: *IEEE Journal of Solid-State Circuits* 48.4 (2013), pp. 940–947.

[RK10] Kurt Rosenfeld and Ramesh Karri. *Attacks and Defenses for JTAG*. In: *IEEE Design & Test of Computers* 27.1 (2010).

[Rüh10] Ulrich Rührmair. *Oblivious Transfer Based on Physical Unclonable Functions*. In: *International Conference on Trust and Trustworthy Computing*. 2010, pp. 430–440.

[RBK10] Ulrich Rührmair, Heike Busch, and Stefan Katzenbeisser. *Strong PUFs: Models, Constructions, and Security Proofs*. In: *Towards Hardware-intrinsic Security*. Springer, 2010, pp. 79–96.

[RD13] Ulrich Rührmair and Marten van Dijk. *On the Practical Use of Physical Unclonable Functions in Oblivious Transfer and Bit Commitment Protocols*. In: *Journal of Cryptographic Engineering* 3.1 (2013), pp. 17–28.

[Rüh+10] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. *Modeling Attacks on Physical Unclonable Functions*. In: *Proceedings of the 17th Acm Conference on Computer and Communications Security*. 2010, pp. 237–249.

[RSS09] Ulrich Rührmair, Jan Sölter, and Frank Sehnke. *On the Foundations of Physical Unclonable Functions.* In: *IACR Cryptology ePrint Archive* (2009), p. 277.

[Rya13] Mike Ryan. *Bluetooth: With Low Energy Comes Low Security.* In: *Proceedings of the 7th Usenix Conference on Offensive Technologies*. 2013.

[SVW10] Ahmad-Reza Sadeghi, Ivan Visconti, and Christian Wachsmann. *Enhancing RFID security and privacy by physically unclonable functions*. In: *Towards Hardware-intrinsic Security*. Springer, 2010, pp. 281–305.

[SWW15] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. *Security and privacy challenges in industrial internet of things*. In: *52nd Acm/edac/ieee Design Automation Conference.* 2015, pp. 1–6.

[SRS04] Junichiro Saito, Jae-Cheol Ryou, and Kouichi Sakurai. *Enhancing Privacy of Universal Re-encryption Scheme for RFID Tags*. In: *Embedded and Ubiquitous Computing: International Conference Euc*. Vol. 4. 2004, pp. 879–890.

[SWE02] Sanjay E Sarma, Stephen A Weis, and Daiel W Engels. *Radio-Frequency Identification Systems*. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Vol. 2. 2002, pp. 454–469.

[Sch+14] André Schaller, Tolga Arul, Vincent van der Leest, and Stefan Katzenbeisser. *Lightweight Anti-counterfeiting Solution for Low-End Commodity Hardware Using Inherent PUFs*. In: *Trust and Trustworthy Computing*. 2014, pp. 83–100.

[SŠK15] André Schaller, Boris Škorić, and Stefan Katzenbeisser. *On the Systematic Drift of Physically Unclonable Functions due to Aging*. In: *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*. 2015, pp. 15–20.

[Sch+17a] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Intrinsic Rowhammer PUFs: Leveraging the Rowhammer Effect for Improved Security*. In: *IEEE International Symposium on Hardware Oriented Security and Trust.* 2017, pp. 1–7.

[Sch14a] Schneier on Security: Security Risks of Embedded Systems. Last accessed on June 16th 2017. 2014. URL: `https://www.schneier.com/blog/archives/2014/01/security_risks_9.html`.

[Sch14b] Bruce Schneier. *The Internet of Things Is Wildly Insecure—And Often Unpatchable*. In: *Wired, Jan* (2014).

[SL12] Geert-Jan Schrijen and Vincent van der Leest. *Comparative Analysis of SRAM Memories used as PUF Primitives*. In: *Conference on Design, Automation and Test in Europe*. 2012, pp. 1319–1324.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. *DRAM Errors in the Wild: a Large-Scale Field Study*. In: *Acm Sigmetrics Performance Evaluation Review*. Vol. 37. 1. 2009, pp. 193–204.

[SSW11] Steffen Schulz, Ahmad-Reza Sadeghi, and Christian Wachsmann. *Short Paper: Lightweight Remote Attestation Using Physical Sunctions*. In: *Proceedings of the Fourth Acm Conference on Wireless Network Security*. 2011, pp. 109–114.

[Sch+17b] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. In: *arXiv preprint arXiv:1702.08719* (2017).

[SD15] Mark Seaborn and Thomas Dullien. *Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges*. In: *Black Hat* (2015).

[Sel+11] Georgios Selimis, Mario Konijnenburg, Maryam Ashouei, Jos Huisken, Harmke de Groot, Vincent van der Leest, Geert-Jan Schrijen, Marten van Hulst, and Pim Tuyls. *Evaluation of 90nm 6T-SRAM as Physical Unclonable Function for secure key generation in wireless sensor nodes*. In: *IEEE International Symposium on Circuits and Systems.* 2011, pp. 567–570.

[Ses+06] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. *Scuba: Secure Code Update by Attestation in Sensor Networks*. In: *Acm Workshop on Wireless Security*. 2006, pp. 85–94.

[Ses+05] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. *PIONEER: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems*. In: *Operating Systems Review* 39.5 (2005), pp. 1–16.

[Ses+04] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. *SWATT: Software-Based Attestation for Embedded Devices*. In: *Security & Privacy*. 2004, pp. 272–282.

[SW05] Yaniv Shaked and Avishai Wool. *Cracking the Bluetooth PIN*. In: *Mobile Systems, Applications, and Services*. 2005, pp. 39–50.

[Sha48] Claude E. Shannon. *A Mathematical Theory of Communication*. In: *The Bell System Technical Journal* 27.4 (1948), pp. 623–656. DOI: `10.1002/j.1538-7305.1948.tb00917.x`.

[SSL12] Peter Simons, Erik van der Sluis, and Vincent van der Leest. *Buskeeper PUFs, a Promising Alternative to D Flip-Flop PUFs*. In: *IEEE International Symposium on Hardware-oriented Security and Trust.* 2012, pp. 7–12.

[Sko12] Sergei Skorobogatov. *Physical Attacks and Tamper Resistance*. In: *Introduction to Hardware Security and Trust*. Springer, 2012, pp. 143–173.

[Smi10] Eric Smith. *iPhone Applications & Privacy Issues: An Analysis of Application Transmission of iPhone Unique Device Identifiers (UDIDs)*. Last accessed on October 10th 2017. 2010. URL: `http://www.pskl.us/wp/wp-content/uploads/2010/09/iPhone-Applications-Privacy-Issues.pdf`.

[STM16] STMicroelectronics. *Proprietary Code Read-Out Protection on Microcontrollers of the STM32L4 Series*. Last accessed on June 23th 2017. 2016. URL: `http://www.st.com/resource/en/application_note/dm00186528.pdf`.

[STM17] STMicroelectronics. *ST STM32F100 Value Line*. Last accessed on May 17th 2017. 2017. URL: `http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1031/LN775`.

[SHO07] Ying Su, Jeremy Holleman, and Brian Otis. *A 1.6 pJ/bit 96% stable chip-ID generating circuit using process variations*. In: *IEEE International Solid-state Circuits Conference*. 2007, pp. 406–611.

[SD07] G Edward Suh and Srinivas Devadas. *Physical Unclonable Functions for Device Authentication and Secret Key Generation*. In: *Design Automation Conference*. 2007, pp. 9–14.

[Sye+12] Bilal Syed, Arpan Pal, Krishnan Srinivasarengan, and P Balamuralidhar. *A Smart Transport Application of Cyber-Physical Systems: Road Surface Monitoring with Mobile Devices*. In: *Sixth International Conference on Sensing Technology*. 2012, pp. 8–12.

[Teh+15] Fatemeh Tehranipoor, Nima Karimian, Kan Xiao, and John Chandy. *DRAM Based Intrinsic Physical Unclonable Functions for System Level Security*. In: *Proceedings of the 25th Edition on Great Lakes Symposium on Vlsi*. 2015, pp. 15–20.

[TW11] Mohammad Tehranipoor and Cliff Wang. *Introduction to Hardware Security and Trust*. Springer Science & Business Media, 2011.

[Tex14] Texas Instruments. *Stellaris LM4F120 LaunchPad Evaluation Kit*. `http://www.ti.com/tool/ek-lm4f120xl`. Last accessed 19th April 2017. 2014. URL: `http://www.ti.com/tool/ek-lm4f120xl`.

[Tex15] Texas Instruments. *Crypto-Bootloader (CryptoBSL) for MSP430FR59xx and MSP430-FR69xx MCUs*. Last accessed on June 23th 2017. 2015. URL: `http://www.ti.com/lit/pdf/slau657`.

[Tex17a] Texas Instruments. *MSP430F5308*. Last accessed on May 17th 2017. 2017. URL: `http://www.ti.com/product/msp430f5308`.

[Tex17b] Texas Instruments . *PandaBoard Platform*. Last accessed on May 17th 2017. 2017. URL: `http://www.ti.com/devnet/docs/catalog/endequipmentproductfolder.tsp?actionPerformed=productFolder&productId=16961`.

[Tru11] Trusted Computing Group. *TPM Main Specification*. Last accessed on April 19th 2017. 2011. URL: `http://www.trustedcomputinggroup.org/resources/tpm%5C_main%5C_specification`.

[Tuy+07] Pim Tuyls, Geert-Jan Schrijen, Frans Willems, Tanya Ignatenko, and B Skoric. *Secure Key Storage with PUFs*. In: *Security with Noisy Data-On Private Biometrics, Secure Key Storage and Anti-Counterfeiting* (2007), pp. 269–292.

[TŠ06] Pim Tuyls and Boris Škorić. *Secret Key Generation from Classical Physics: Physical Uncloneable Functions*. In: *Amiware Hardware Technology Drivers of Ambient Intelligence*. Springer, 2006, pp. 421–447.

[V+03] István Vajda, Levente Buttyán, et al. *Lightweight Authentication Protocols for Low-Cost RFID Tags*. In: *Second Workshop on Security in Ubiquitous Computing–ubicomp*. Vol. 2003. 2003.

[VBN15] Pol Van Aubel, Daniel J Bernstein, and Ruben Niederhagen. *Investigating SRAM PUFs in Large CPUs and GPUs*. In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. 2015, pp. 228–247.

[VSL13] Robbert Van Den Berg, Boris Skoric, and Vincent van der Leest. *Bias-Based Modeling and Entropy Analysis of PUFs*. In: *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*. 2013, pp. 13–20.

[Van+10] Vincent Van der Leest, Geert-Jan Schrijen, Helena Handschuh, and Pim Tuyls. *Hardware Intrinsic Security from D Flip-Flops*. In: *Proceedings of the Fifth Acm Workshop on Scalable Trusted Computing*. 2010, pp. 53–62.

[Van15] Anthony Van Herrewege. *Lightweight PUF-based Key and Random Number Generation*. PhD thesis. 2015.

[Van+12] Anthony Van Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. *Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-Enabled RFIDs*. In: *International Conference on Financial Cryptography and Data Security*. 2012, pp. 374–389.

[Van+13] Anthony Van Herrewege, Vincent van der Leest, André Schaller, Stefan Katzenbeisser, and Ingrid Verbauwhede. *Secure PRNG Seeding on Commercial Off-The-Shelf Microcontrollers*. In: *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*. 2013, pp. 55–64.

[VDP16] Elena Ioana Vatajelu, Giorgio Di Natale, and Paolo Prinetto. *Towards a Highly Reliable SRAM-Based PUFs*. In: *Design, Automation & Test in Europe Conference & Exhibition*. 2016, pp. 273–276.

[Vee+16] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. *Drammer: Deterministic Rowhammer Attacks on Mobile Platforms*. In: *Proceedings of the 2016 Acm Sigsac Conference on Computer and Communications Security*. 2016, pp. 1675–1689.

[VT12] John Viega and Hugh Thompson. *The State of Embedded-Device Security (Spoiler Alert: It's Bad)*. In: *IEEE Security & Privacy* 5 (2012), pp. 68–70.

[Wan+12] Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G Edward Suh, and Edwin C Kan. *Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints*. In: *IEEE Symposium on Security and Privacy*. 2012, pp. 33–47.

[Wei+04] Stephen A Weis, Sanjay E Sarma, Ronald L Rivest, and Daniel W Engels. *Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems*. In: *Security in Pervasive Computing*. Springer, 2004, pp. 201–212.

[Wik17] Wikipedia: DeCSS. Last accessed on June 16th 2017. 2017. URL: `http://en.wikipedia.org/wiki/DeCSS`.

[WST95] Frans MJ Willems, Yuri M Shtarkov, and Tjalling J Tjalkens. *The Context-Tree Weighting Method: Basic Properties*. In: *IEEE Transactions on Information Theory* 41.3 (1995), pp. 653–664.

[Xia+14] Kan Xiao, Md Tauhidur Rahman, Domenic Forte, Yu Huang, Mei Su, and Mohammad Tehranipoor. *Bit Selection Algorithm Suitable for High-Volume Production of SRAM-PUF*. In: *IEEE International Symposium on Hardware-oriented Security and Trust.* 2014, pp. 101–106.

[Xia+16] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. *One Bit Flips, One Cloud Flops: Cross-vm Row Hammer Attacks and Privilege Escalation*. In: *25th USENIX Security Symposium.* 2016, pp. 19–35.

[Xio+16] Wenjie Xiong, André Schaller, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Run-time Accessible DRAM PUFs in Commodity Devices*. In: *International Conference on Cryptographic Hardware and Embedded Systems*. 2016, pp. 432–453.

[Yos+06] Akito Yoshida, Jun Taniguchi, Katsumasa Murata, Morihiro Kada, Yusuke Yamamoto, Yoshinori Takagi, Takeru Notomi, and Asako Fujita. *A Study on Package Stacking Process for Package-on-Package (PoP)*. In: *Proceedings of the 56th Electronic Components and Technology Conference*. 2006, 6–pp.

[Zan+14] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. *Internet of Things for Smart Cities*. In: *IEEE Internet of Things journal* 1.1 (2014), pp. 22–32.

[Zue08] Detlef Zuehlke. *Smartfactory–From Vision to Reality in Factory Technologies*. In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 14101–14108.

[Zue10] Detlef Zuehlke. *SmartFactory—Towards a Factory-of-Things*. In: *Annual Reviews in Control* 34.1 (2010), pp. 129–138.

Chapter A

List of Publications

Journal Articles

[Sch+17a] André Schaller, Taras Stanko, Boris Škorić, and Stefan Katzenbeisser. *Eliminating Leakage in Reverse Fuzzy Extractors*. In: *IEEE Transactions on Information Forensics and Security* (2017).

[Sch+17b] André Schaller, Wenjie Xiong, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Škorić, Stefan Katzenbeisser, and Jakub Szefer. *Decay-Based DRAM PUFs in Commodity Devices*. In: *IEEE Transactions on Dependable and Secure Computing* (2017). Submitted.

[KS12] Stefan Katzenbeisser and André Schaller. *Physical Unclonable Functions*. In: *Datenschutz und Datensicherheit-DuD* 36.12 (2012), pp. 881–885.

Conference Papers

[Sch+17c] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Intrinsic Rowhammer PUFs: Leveraging the Rowhammer Effect for Improved Security*. In: *IEEE International Symposium on Hardware Oriented Security and Trust*. 2017, pp. 1–7.

[Sch+17d] Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. *Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors*. In: *European Symposium on Research in Computer Security*. 2017, pp. 437–455.

[Xio+16] Wenjie Xiong, André Schaller, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. *Run-time Accessible DRAM PUFs in Commodity Devices*. In: *International Conference on Cryptographic Hardware and Embedded Systems*. 2016, pp. 432–453.

[KSK15] Florian Kohnhäuser, André Schaller, and Stefan Katzenbeisser. *PUF-Based Software Protection for Low-End Embedded Devices*. In: *Trust and Trustworthy Computing*. 2015, pp. 3–21.

[Sch+14] André Schaller, Tolga Arul, Vincent van der Leest, and Stefan Katzenbeisser. *Lightweight Anti-counterfeiting Solution for Low-End Commodity Hardware Using Inherent PUFs*. In: *Trust and Trustworthy Computing*. 2014, pp. 83–100.

Workshop Papers and Demos

[SŠK15] André Schaller, Boris Škorić, and Stefan Katzenbeisser. *On the Systematic Drift of Physically Unclonable Functions due to Aging*. In: *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*. 2015, pp. 15–20.

[SL13] André Schaller and Vincent van der Leest. *Physically Unclonable Functions Found in Standard Components of Commercial Devices*. In: *First Workshop on Trustworthy Manufacturing and Utilization of Secure Devices (trudevice), Avignon, France*. 2013, pp. 1–2.

[Van+13a] Anthony Van Herrewege, Vincent van der Leest, André Schaller, Stefan Katzenbeisser, and Ingrid Verbauwhede. *Secure PRNG Seeding on Commercial Off-The-Shelf Microcontrollers*. In: *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*. 2013, pp. 55–64.

[Van+13b] Anthony Van Herrewege, André Schaller, Stefan Katzenbeisser, and Ingrid Verbauwhede. *DEMO: Inherent PUFs and Secure PRNGs on Commercial Off-The-Shelf Microcontrollers*. In: *Proceedings of the 2013 Acm Sigsac Conference on Computer & Communications Security*. 2013, pp. 1333–1336.

Non-Peer-Reviewed Publications

[Ana+17] Nikolaos Athanasios Anagnostopoulos, André Schaller, Yufan Fan, Wenjie Xiong, Fatemeh Tehranipoor, Tolga Arul, Sebastian Gabmeyer, Jakub Szefer, John A Chandy, and Stefan Katzenbeisser. *Insights into the Potential Usage of the Initial Values of DRAM Arrays of Commercial Off-the-Shelf Devices for Security Applications*. In: *26 th Crypto-Day* (2017), p. 1.

[Ana+16] Nikolaos Anagnostopoulos, Stefan Katzenbeisser, Markus Rosenstihl, André Schaller, Sebastian Gabmeyer, and Tolga Arul. *Low-Temperature Data Remanence Attacks Against Intrinsic SRAM PUFs.* In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 769.