# PUFFIN

## Physically unclonable functions found in standard PC components

Project number: 284833
FP7-ICT-2011-C

# D3.1

## Scientific contribution of WP3, part 1 Use Cases

Due date of deliverable: 31. July 2013
Actual submission date: 30. September 2013

WP contributing to the deliverable: WP3

Start date of project: 1. February 2012 Duration: 3 years

Coordinator:
Technische Universiteit Eindhoven
Email: `coordinator@puffin.eu.org`
`www.puffin.eu.org`

Revision 1.0

| **Project co-funded by the European Commission within the 7th Framework Programme** | | |
|---|---|---|
| **Dissemination Level** | | |
| **PU** | Public | X |
| **PP** | Restricted to other programme participants (including the Commission services) | |
| **RE** | Restricted to a group specified by the consortium (including the Commission services) | |
| **CO** | Confidential, only for members of the consortium (including the Commission services) | |

# Scientific contribution of WP3, part 1
# Use Cases

André Schaller (TUD)
Stefan Katzenbeisser (TUD)
Vincent van der Leest (IID)
Ruben Niederhagen (TUE)

30. September 2013
Revision 1.0

The work described in this report has in part been supported by the Commission of the European Communities through the FP7 program under project number 284833. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

## Abstract

This document contains an overview results obtained within Work Package 3 (WP3) of the PUFFIN project. The work in WP3 can be divided into three tasks: 1.) development of hardware-entangled cryptographic building blocks, which are based on the underlying PUFs; 2.) exploration of the applicability of the PUF instances found in WP2 to be used as a TPM replacement; 3.) investigation of approaches to bind software to a platform by intertwining PUF responses with the binary running on the hardware. This document describes the achievements in these areas during the first phase (18 months) of the PUFFIN project.

**Keywords:** WP3, use cases, PUF protocols, proof-of-concepts

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Work Package 3 (WP3) is devoted to the development of use cases and applications for the PUF instances acquired in Work Package 1 and 2 of the PUFFIN project. Specifically, to employ intrinsic PUFs as a hardware-based anchor of trust, WP3 investigates the following aspects:

- Task 3.1 – Hardware-based trust establishment.
- Task 3.2 – PUF-based software security.
- Task 3.3 – Security evaluation and implementation.

On the next pages the results of previous research and current work in progress are summarized. Furthermore, future work as well as scheduled projects are presented. The architecture of the PUFFIN project implies a bottom-up relationship between the different work packages. Thus, the work of WP3 is based on the located and qualified PUF instances from WP1 and WP2. The main contribution of WP3 during the first phase of the PUFFIN project consists of the development of use cases and applications for intrinsic PUFs, whose requirements towards the PUF-quality was fed back to WP1 and WP2. Furthermore, initial implementations of the proposed use cases are provided, which will be advanced and extended in the second half of the PUFFIN project.

Chapter 2 presents use cases which were gathered specifically for intrinsic PUF instances. This collection is rather abstract and extensive and was used as a basis within the project for the actual development of selected use cases.

Chapter 3 presents selected use cases which were further evaluated and implemented. We present an improved Helper Data scheme based on a modified Code Offset Method, which is part of Task 3.1. The idea is to achieve Zero Secrecy Leakage Security (ZSL) of the Helper Data in Fuzzy Extractors given non-uniformly distributed PUF measurements. The desired ZSL property is established by introducing bogus Helper Data instances to the publicly stored enrollment data. Furthermore, we will demonstrate an implementation of a secure boot approach using intrinsic SRAM PUFs on a system-on-a-chip (SoC) platform, covering Task 3.2. The idea is to bind a given software instance to the hardware it is supposed to be running on. In contrast to traditional schemes, which mostly involve Trusted Platform Modules (TPMs) and thus require additional hardware modification and key and signature management overhead, we propose to employ intrinsic SRAM PUFs as the hardware-based anchor of trust. As a second contribution to Task 3.1, we demonstrate the concept of using the inherent noise of static random-access memory (SRAM) start up patterns as a source of true random numbers. We examined the on-board SRAM of two wide-spread micro-controllers and developed an architecture which can be used to provide a stream of random number of variable length. Thus, we show the possibility to provide a light-weight source of high-quality random numbers in commercial off-the-shelf (COTS) devices, which can be used as input for subsequent cryptographic primitives and protocols. During the second period of the PUFFIN project, current research results will be used as a basis to create more sophisticated approaches to hardware-software binding as well as approaches to the protection of intrinsic PUFs from passive listening and emulation attacks. Furthermore, a goal of the second period is the development of novel cryptographic protocols based on intrinsic PUFs. Later, the proposed protocols and implemented solutions will be tested towards their security. Thus, the second period of the PUFFIN project will cover Task 3.3 by analyzing the security of the research results of WP3.

# Chapter 2

# Use Cases

## 2.1 Introduction

One of the initial goals of WP3 is to explore possible use cases for intrinsic PUF instances which were already discovered and which were anticipated to be available in the near future of the PUFFIN project. The challenge is to consider the characteristics of intrinsic PUFs in particular with respect to aspects of security. To summarize, the security characteristics of intrinsic PUFs essentially comprise

1. a small challenge-response-pair, which might enable an attacker to perform Man-in-the-Middle attacks or to model the PUF itself, and
2. the public availability of the PUF instance.

The second challenge comes into place since the project is on exploring PUF instances in commodity hardware, which is supposed to be accessible also to a maliciously party. Furthermore, as the goal of WP3 is to deliver a working proof-of-concept of one of the proposed security architectures involving PUFs, the collection of use cases shall have a pragmatic character. The rest of this chapter includes a brief overview of the selected use cases, which we will include in further research of the PUFFIN project.

Table 2.1 shows all the proposed use cases and gives an indication on whether we selected them for further research. The research process for the selected use cases spans over the entire project time. The last columns shows in which period of the PUFFIN project the corresponding use cases are subject of research. Thus, WP3 research of period one of the PUFFIN project focuses on a subset of the selected use cases. Research results of this subset are presented in Section 3. Use cases for which no research or development results are given will be processed in period two of the PUFFIN project.

## 2.2 Random Number Generation

The idea of this use case is diametrical to the traditional use of PUFs as a fingerprint for silicon chips. For the extraction of hardware identifiers which can be regarded as a fingerprint of the chip, only the stable bits are considered to generate a cryptographic key. However, in case of random number generation the more interesting bits are the unstable ones which are inherent to any PUF response and which are usually referred to as noise. Since this noise

Table 2.1: Proposed use cases for intrinsic PUFs.

| Use Case | Further Research | Dedicated Period |
|---|---|---|
| Random Number Generation | yes | 1 |
| Key Storage | yes | 1 & 2 |
| Authentication | yes | 2 |
| Hardware-Software-Binding / Secure Boot | yes | 1 & 2 |
| Remote Service Activation | no | — |
| Key Distribution | no | — |
| Tamper Evidence | no | — |

completely depends upon the physical hardware characteristics of the underlying platform, they are a worthwhile source for true random numbers. These unstable bits are present only directly after the device is booted, because they are likely to be overwritten soon by the operation system or by application data. Therefore, they need to be made available for further processing right after boot time: The initial true random numbers extracted from the noise in the SRAM patterns are used as a seed for a pseudo-random number generator (PRNG) to provide a bit stream of random numbers of variable length. However, since we need to assume that an adversary has access to the device, provisions need to be made to ensure that the initially extracted true random numbers are not disclosed publicly. Thus, the seed must to be put into the PRNG as early as possible. Furthermore, the initial startup values of the SRAM must be overwritten to disable any unintended posterior access.

## 2.3 Key Storage

Key storage is the most obvious but also the most demanded use case since it provides the basis for many of the other use cases. In contrast to traditional approaches to store cryptographic keys where the keys are present in non-volatile memory (NVM) all the time, using PUFs the keys are generated on-the-fly. Since the keying material is present only during a short time span while the device is operating, the attack surface for invasive attacks is decreased. Again, since the device is assumed to be publicly available, precautions have to be implemented to shield the key generation function from public usage.

## 2.4 Authentication

Based on the use case of generating keys from PUF responses, these keys can be further used for performing authentication between different parties. Here, we can construct three different variants of the use case:

1. device authentication,
2. client authentication, and
3. server authentication.

Regarding device authentication, a given hardware wants to authenticate itself against a user. It is assumed that the device is light-weight in the sense of low computational power. In this case a smaller challenge-response-pair (CRP) space is not much of a security issue since it can be assumed that there is no possibility of an adversary to position himself between device and user.

The client authentication use case shares the same basic idea as the device authentication except that here the client wants to authenticate himself to a remote server. Again, the client is assumed to be light-weight. In contrast the remote server is assumed to have more computational power. In this scenario the small CRP space must be considered since both parties are assumed to communicate over an insecure channel making it possible for an adversary to eavesdrop the exchanged message. Thus, the attacker might be able to perform a Man-in-the-Middle attack or capture enough messages to simulate the PUF instance.

The server authentication scenario shares most aspects from the client authentication use case. Here, typically the server holds the PUF instance while the client checks the server's authenticity by querying its CRP database.

## 2.5 Hardware-Software-Binding / Secure Boot

The goal of this use case is to bind a given software instance to a platform, i.e., the software will only behave properly if it is executed on the correct hardware. Since the hardware is assumed to integrate a PUF instance, the binary needs to be intertwined with the PUF responses.

A first approach uses the PUF to generate a key on-the-fly, which is subsequently used to decrypt the bootloader which handles the loading of the kernel and thus of the operating system. In this case the initial key-generation as well as the decrypting mechanism must be made irreversible such that an adversary is not able to exchange it and thus bypass it. By installing the critical parts of the code into masked ROM this requirement is guaranteed and a defined, secure state at bootloader time can be achieved.

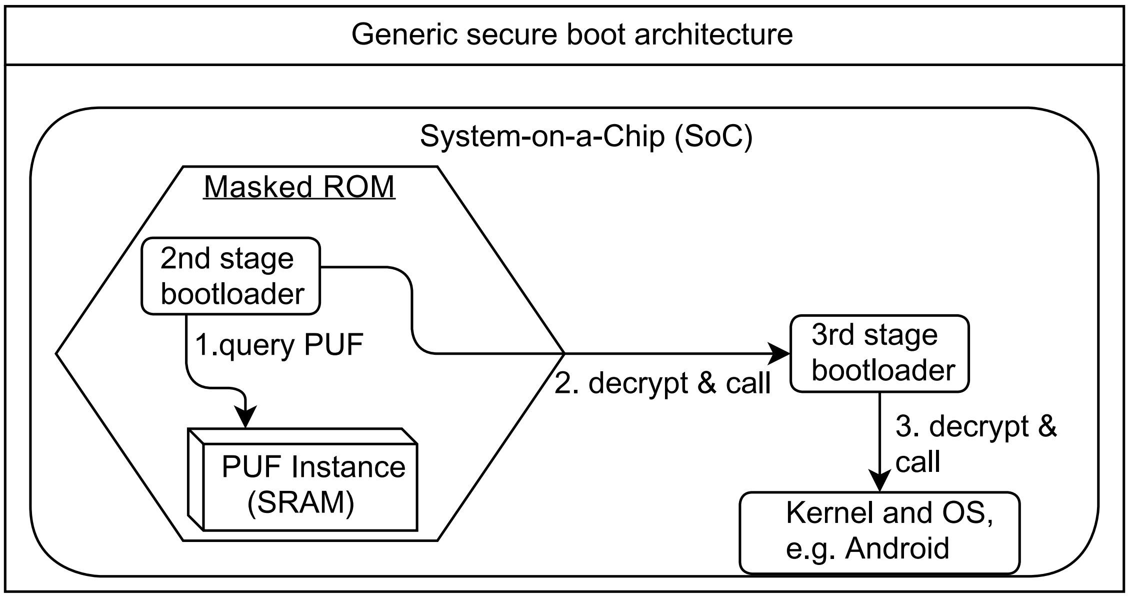

Figure 2.1: Scheme of the generic secure boot architecture.

In the second phase of the project we focus on a more advanced approach which intertwines the binary and the PUF responses more tightly. The idea is to challenge the PUF during the execution of the binary such that the binary only continues to execute properly if it receives the matching PUF responses; if it is executed on the wrong hardware, the software becomes buggy. This approach is not straight-forward and requires several security mechanisms like obfuscation and white-box cryptography. However, in this case the small size of the CRP space becomes an issue that needs to be addressed, because an attacker is able to run the software in an virtual environment to capture the challenge-response communication between hard- and software and thus is able to model the PUF.

# Chapter 3

# Development of Use Cases

## 3.1 Introduction

In this chapter we present results of the selected use cases, which were subject to research for WP3 during the first period of the PUFFIN project. The presented research projects stem from the use case collection, which was presented in chapter 2. After selecting interesting use cases from the collection for further research, a finer selection was made regarding, which project to process in the first period of the PUFFIN project and which in the second one. The selected research projects for the first period of the PUFFIN project, which are presented in the following, cover the following tasks of WP3:

- Task 3.1: Hardware-based trust establishment.
- Task 3.2: PUF-based software security.

The work of Boris Skoric and Niels de Vreede is devoted to Task 3.1 as it introduces a novel Helper Data Scheme, which realizes a new type of Zero Secrecy Leakage. Task 3.2 is covered by two contributions. The first work of Anthony Herreweg et. al. implements a secure PRNG using intrinsic PUFs, whilst the second contribution from André Schaller et. al. uses intrinsic PUFs to securely bind a software instance, in particular a mobile device bootloader, to a hardware platform.

In the second period of WP3, these results will be used as a basis to improve current work and create more sophisticated solutions. These will address the most challenging part of Task 3.2 to create an approach, which protects intrinsic PUFs from emulation and replay attacks. Furthermore, a more advanced approach for intertwining PUF challenges and responses with a given software binary will be adressed. Another goal of WP3 during the second period is to establish new cryptographic protocols based on the hardware properties of the underlying intrinsic PUFs. The latter part of the second period of the PUFFIN project is devoted to test and analyze the security of the proposed protocols and implementations. Thus, Task 3.3 – security evaluation and implementation – will be covered during this phase of the PUFFIN project by WP3.

## 3.2 Helper Data Schemes

Using Physically Unclonable Functions, such as a secure key storage or for other cryptographic targeted applications, the basic principle always is to reconstruct a secret $S$ from several noisy measurements $X_1, X_2, ..., X_n$. To achieve a reliable reconstruction given a set of measurements with a known and bounded bit error rate (BER), so-called Fuzzy Extractors (FEs) are employed. FEs are a specialized type of a security primitive, referred to as Helper Data Scheme (HDS). In general HDS operate in two phases: (1) the enrollment phase and (2) the reconstruction phase. In the enrollment phase a reference measurement $X$ of a given PUF instance is used as input (and optionally a random value $R$) for the FE, which subsequently outputs a secret $S$ as well as Helper Data $W$. In the reconstruction phase, the Helper Data $W$ is used to reconstruct $S$ using a different but similar noisy measurement $X'$.

The HDS and hence the FE are constructed such that the Helper Data generated during the enrollment phase, does not leak too much information about the secret $S$, such that $W$ can be stored publicly. This security property is also referred to as Zero Secrecy Leakage (ZSL). However, using the Code Offset Method (COM) as one method to construct a FE, the security property of ZSL only holds if the measurements $X_1, X_2, ..., X_n$ are uniformly distributed.

The paper presented in Appendix B.1 deals with a modified Code Offset Method to achieve Zero Secrecy Leakage property even if the involved measurements are not uniformly distributed. The basic approach adds bogus Helper Data instances to the publicly stored enrollment data, which are randomly permuted. From the attacker point of view it is hard to distinguish between real and fake Helper Data Instances in this permuted set. In contrast, the legit user possesses additional information about the permutation and can easily recover relevant elements from the permuted set. If this set has enough fake elements an attacker can only use brute force, which will not be longer feasibly as soon as the number of fake elements reaches a certain value. In this way a new type of Zero Secrecy Leakage is achieved distinct from the traditional approach to ZSL property.

The analysis of the proposed method reveals that for a small spam factor $m$, the workload for the adversary increases by $\log m$ bits. For large $m$, the leakage $I(X;W)$ is practically eliminated. Whilst the workload for the adversary is increased, the workload for the legit party stays almost constant as a function of $m$.

## 3.3 Light-Weight Secure Boot

This paper presents a light-weight secure boot solution particularly suitable for mobile commercial off-the-shelf hardware such as mobile phones and tablet PCs. Secure boot is enabled by employing intrinsic Physically Uncloneable Functions, which are derived from the process variations of static random-access memory in these devices. In contrast to the traditional secure boot approach employing TPMs, our solution does not need additional hardware, making it more flexible and cost efficient. We evaluated our approach through an implementation on a System-on-a-Chip (SoC) platform. We selected a SoC architecture as most of the modern smartphones and tables are designed as such and the increasing usage of SoCs in mobile devices makes it likely that future devices will be designed using the same principle. We selected the PandaBoard as basis for our implementation. The PandaBoard (ES) is an OMAP4430 (4460) based platform comprising of two ARM Cortex-A9 as well as two Cortex-M3 for signal processing. On basis of this hardware configuration we consider this to be an adequate reference setup since many of current smartphones show a similar ARM-based configuration. Next to the external 2 GiB DDR memory it consists of several on-chip memory (OCM) instances. Analysis of the OCM instances revealed, that only some portions of the RAM shows PUF-like behavior after startup. In particular the Level-3 OCM RAM can be partially used to extract a fingerprint for a given PandaBoard. The Level 3 OCM RAM (L3 RAM) consists of 56 KiB volatile on-Chip RAM.

Figure 3.1 shows the bitmap of the whole L3 RAM from a PandaBoard shortly after the device gets out of reset.

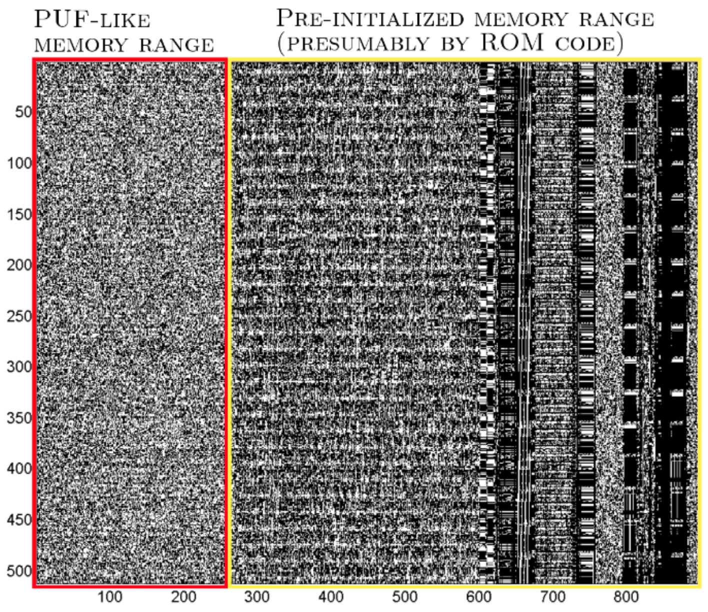

Figure 3.1: Bit map of a single enrollment of the L3 OCM RAM from a PandaBoard. The red area can be used for extracting a fingerprint. The yellow area consists of initialized values and thus does not show PUF characteristics.

In this project we use u-boot, which is one of the most widely used bootloaders. It integrates a second-stage bootloader (Memory Locator – *MLO*), which is small enough to fit into internal memory, and a third-stage bootloader (*u-boot.img*). The *MLO* performs some hardware initialization as well as the setup of external memory. Afterwards it calls the *u-boot.img*), which is copied to external memory, and initializes further hardware components and eventually calls the operating system kernel. A more detailed scheme of the enrollment as well as the reconstruction phase of our proof-of-concept is depicted in Figure 3.2 and Figure 3.3.

The evaluation of the PUF characteristics were performed on five different PandaBoards with 1000 measurements for each device. We conducted analysis on the Hamming Weight, the within-class Hamming distance as well as the between-class Hamming distance. The results for all the metrics showed almost ideal characteristics, which can be looked up in detail in the original paper (see Appendix A.1).

## 3.4 Secure PRNG Seeding

This project deals with the problem of weak seeds used to initialize PRNGs. Such weak seeds lead to the PRNG generating predictable random numbers. The project presents a lightweight

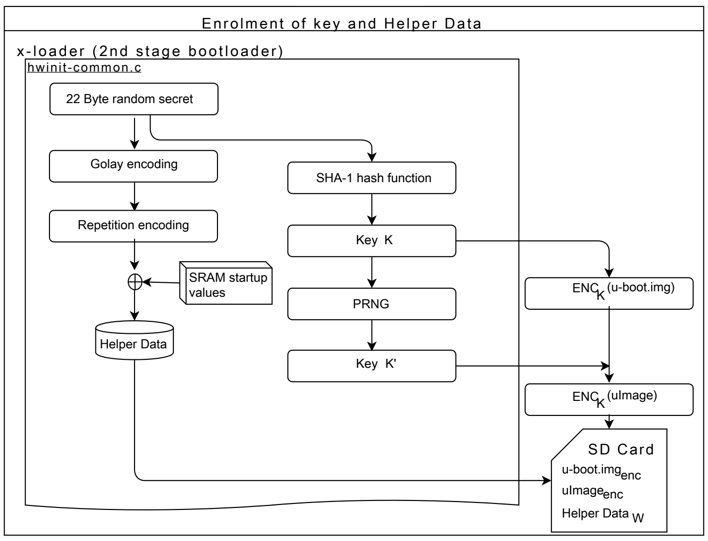

Figure 3.2: Detailed scheme of the enrollment phase of the implementation. The enrollment is performed at the manufacturer's site and needs to be conducted once per device.

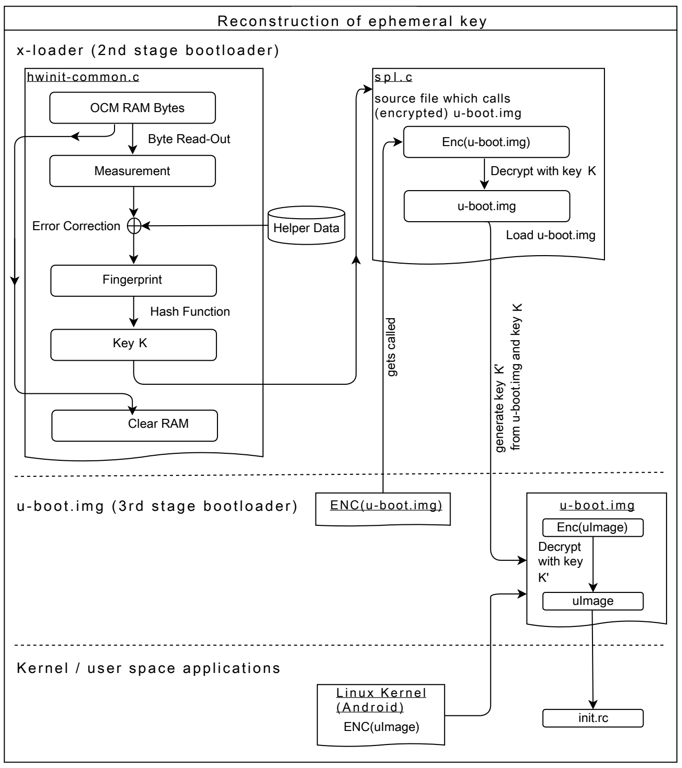

Figure 3.3: The reconstruction phase of the implementation restores a secret key from several noisy measurements by means of publicly stored Helper Data.

software-only approach to generate secure seeds on commercial off-the-shelf microcontrollers. After identifying and evaluating SRAM in commercial off-the-shelf microcontrollers as an entropy source for PRNG seeding, the start-up patterns of two popular types of microcontrollers, a STMicroelectronics STM32F100R8 and a Microchip PIC16F1825 were measured and evaluated. Also, an efficient software-only architecture for secure PRNG seeding was implemented. After analyzing over 1 000 000 measurements in total, we conclude that of these two devices, the PIC16F1825 cannot be used to securely seed a PRNG. The STM32F100R8, however, has the ability to generate very strong seeds from the noise in its SRAM start-up pattern. These seeds can then be used to ensure a PRNG generates high quality data. Figure 3.1 depicts the general architecture of the proposed approach.

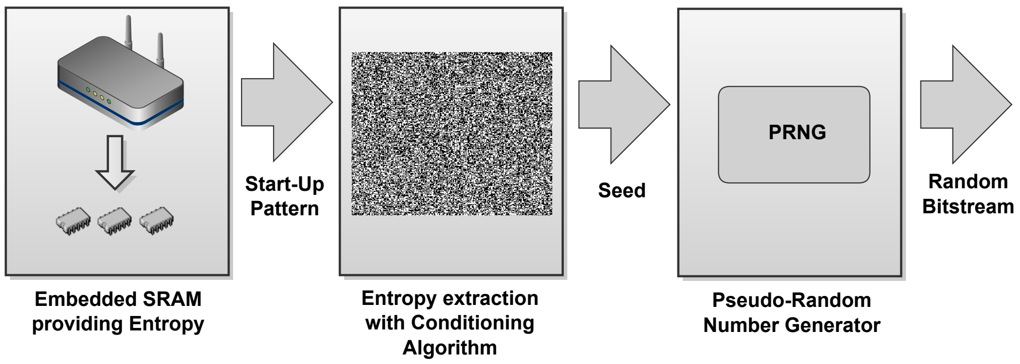
Start-Up Pattern
Seed
PRNG
Random Bitstream
Embedded SRAM providing Entropy
Entropy extraction with Conditioning Algorithm
Pseudo-Random Number Generator

Figure 3.1: Architecture for secure seeding of PRNGs. SRAM start-up patterns are used as a source of entropy, which is subsequently exploited for generating a secure seed.

# Appendix A

# Use Cases

## A.1 Paper: "Light-Weight Secure Boot for SoCs using Physically Unclonable Functions"

**Authors:** André Schaller, Tolga Arul, and Vincent van der Leest (Intrinsic-ID)
**Venue:** Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)
**Date:** November 4th - 8th, 2013

**Status:** Work in progress. A draft of the paper is given below.

# Light-Weight Secure Boot for System-on-Chips using Physically Unclonable Functions

André Schaller
TU Darmstadt
Darmstadt, Germany
schaller@seceng.informatik.tu-darmstadt.de

Tolga Arul
CASED
Darmstadt, Germany
tolga.arul@cased.de

Vincent van der Leest
Intrinsic-ID
Eindhoven, The Netherlands
vincent.van.der.leest@intrinsic-id.com

Stefan Katzenbeisser
TU Darmstadt
Darmstadt, Germany
katzenbeisser@seceng.informatik.tu-darmstadt.de

## ABSTRACT

This paper presents a light-weight secure boot solution particularly suitable for mobile commercial off-the-shelf hardware such as mobile phones and tablet PCs. Secure boot is enabled by employing intrinsic Physically Uncloneable Functions, which are derived from the process variations of static random-access memory in these devices. In contrast to the traditional secure boot approach employing TPMs, our solution does not need additional hardware, making it more flexible and cost efficient. The deployment of the PUF only requires the modification of the bootloader and the presence of components, which are already widely available on the targeted devices. In a proof of concept, our solution is embedded in an state-of-the-art commodity system-on-a-chip platform, whose architecture is common to most current smartphones and tablets. As a result of the subsequent evaluation, we show that the quality of the intrinsic PUF characteristics in our solution is almost ideal.

## Categories and Subject Descriptors

B.3.1 [**Semiconductor Memories**]: SRAM; D.4.6 [**Security and Protection**]: Cryptographic controls

## General Terms

Security

## Keywords

Software binding, Physically Unclonable Functions, Embedded Security, System-on-a-Chip

## 1. INTRODUCTION

With the proliferation of mobile computing (like smartphones and tablets) the influence of mobile devices on our every day communication increases. As the computational power of these devices steadily grows they serve as a platform for various applications, which used to be run on traditional desktop PCs and laptops. Such applications range from business applications over access to social networks to security-critical scenarios like online banking.

However, due to the dissemination and the ongoing trend of employing mobile devices as a full-grown replacement of traditional computing hardware, they are already a worthwhile target of cyber-criminals. Malware targeted at mobile operating systems (e.g. Android or iOS) is being actively developed and already achieved a comparable complexity as their counterparts infecting desktop PCs and laptops [15]. Malware prototypes with simple logic – like Cabir worm targeting Symbian OS [9] – were developed further to actively deployed malware of high complexity [16]. Even advanced malware like Android rootkits [18] exists today. This trend is reflected in a constantly growing number of mobile applications which are identified as behaving maliciously [17].

Due to the rising number of mobile malware as well as the fact that sensitive data is increasingly often stored on and shared among such devices, a trustworthy environment for mobile devices is needed. To ensure an overall protection, a security mechanism needs to be anchored at a low level of the platform architecture of the involved embedded device and preferably rely on a hardware-based anchor of trust. A secure boot approach can achieve these requirements by establishing trust in the operating system that was booted and thus mitigate malicious modifications of the operating system, for example by rootkits. Current approaches such as Trusted Platform Modules provide a solution, yet they require additional hardware modifications which might be uneconomic for low-end devices, impeding a broad commitment to such approaches. Besides the protection of end-users, vendors want to prevent malicious modifications of their firmware and mobile operating systems to protect carrier-specific services and thus their business model from such threats.

In this paper we propose a light-weight secure boot approach, which guarantees a verified system state at boot time and thus provides protection from firmware or kernel modifications by rootkits or other attacks. The proposed solution extracts an identifier from the underlying hardware to generate a cryptographic key at boot time, which is used to bind a software instance to the platform.

### 1.1 Contributions

The proposed approach uses intrinsic Physical Unclonable Functions (PUFs), which are based on the process variations of static random-access memory (SRAM) found in commodity hardware. These PUFs are used to generate an ephemeral cryptographic key during an early boot stage, which is unique for each device and used to authenticate the software running on the platform.

We achieve a light-weight key storage solution which does – in contrast to current TPM approaches – not require the integration of additional hardware. The key generation and decryption mechanism is implemented at the earliest stage of the booting process to decrypt the bootloader and thus build a chain-of-trust to guarantee a verified state at boot time. Since the cryptographic key is generated on the fly, it only exists in memory during the phase of decryption and is never stored permanently. Thus, the attack surface for invasive key extraction attacks is highly decreased. As the cryptographic key is unique for every individual device, it is not possible to provide a generic attack even if adversaries would succeed in deriving the key from a device. Thus, the economic motivation for an adversary is low.

We show how to instantiate the technologies on the PandaBoard, which is a system-on-a-chip (SoC) platform utilizing state-of-the-art ARM-based processors, typifying modern smart phones.

## 2. RELATED WORK

### 2.1 Current developments regarding PUFs

A Physically Unclonable Function (PUF) is a complex physical structure that generates a value $y$ in response to a stimulus $x$. The response $y$ depends on the challenge $x$ as well as on the micro- or nanoscale physical structure of the PUF itself. It is assumed that the PUF is unclonable such that it can not 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 can not be predicted. Furthermore, due to minuscule manufacturing variations during the production process, embedded PUFs can be used to robustly identify a silicon chip.

Silicon-based PUFs can be delay-based or memory-based PUFs. For an exhaustive overview of PUFs and details on their taxonomy we refer to [14]. It has been shown that selected statical random-access memory (SRAM) shows PUF-like behavior [10]. Further research in this area support the applicability of SRAM as a Physical Unclonable Function [13] [12]. Using SRAM as PUFs exploits manufacturing variations which manifest themselves in a bias of memory cells inside of SRAM modules. During the power up phase these cells initialize to either the value of zero or one. Most cells prefer to initialize to one of both values, which in total creates a start-up pattern we will exploit to generate a fingerprint for the device. Since not all of the SRAM bytes show a stable behavior in such sense that they are always initialized to a fixed value, the SRAM start-up values include a small amount of unstable bits, so-called noise. Since the goal is to reconstruct a reliable cryptographic key from several noisy measurements, the noise is eliminated by employing a Fuzzy Extractor [8], which extracts the stable part of the PUF response and transforms it to a uniformly distributed value. The Fuzzy Extractor implements two functions, an enrollment function and a reconstruction function. During the enrollment phase, which ought to be performed by the mobile phone's manufacturer, the Fuzzy Extractor takes a reference measurement $R$ as input and outputs a cryptographic key $K_i$ as well as Helper Data $W_i$. In the reconstruction phase, $K_i$ will be reconstructed out of a noisy measurement with help of $W_i$. The Helper Data can be stored publicly since it does not leak information about $K_i$.

### 2.2 Current approaches to secure boot

Traditional approaches to secure boot involve the usage of a Trusted Platform Module (TPM) or rely on vendor-specific TPM-like security extensions like ARM's TrustZone [2], TI's M-Shield [11] or implementations of secure virtual machines from Intel or AMD. The TPM [6] was proposed by the Trusted Computing Group and constitutes a hardware-based solution with the TPM chip implementing cryptographic primitives such as random number generators, a cryptographic coprocessor and a secure memory. The latter is primarily used to store cryptographic keys and certificates and more. Additionally, TPMs for mobile devices, so-called Mobile Trusted Modules (MTMs), have been developed.

In [4] the authors propose a PUF-based solution to protect cryptographic keys inside the TPM chip from non-invasive attacks as they are sent over the internal bus. They integrate a PUF-instance inside the TPM and use it to encrypt TPM keys before they are sent over the internal bus to mitigate key extraction by probing the bus lines. Even though this approach also employs a PUF instance to generate a key based on the underlying hardware it still requires a TPM chip to realize the actual secure boot.

In this work we focus on memory-based PUFs found in statical RAM in commercial off-the-shelf (COTS) devices. We use the intrinsic SRAM PUF to generate a device-specific cryptographic key at boot time to bind a legit kernel to the platform and thus guarantee a trusted operating system state without additional hardware requirements. To the best of our knowledge this is the first implementation of a secure boot approach, which is completely based on PUFs without requiring additional hardware modifications.

## 3. ARCHITECTURE

### 3.1 General Architecture

The proposed light-weight secure boot solution is designed to be implemented on commodity mobile devices. Our solution only requires hardware components which are already present in virtually any computing device. In particular, we require the mobile devices to be equipped with a masked read-only memory (ROM), a SRAM module and a CPU. Furthermore, we assume that the device is started by a two-staged booting process, involving a second-stage bootloader as well as a third-stage bootloader[1]. This multipart boot process is common for embedded devices as well as for x86 processors (e.g. using GRUB bootloader).

Our design employs an second-stage bootloader and an encrypted 3rd-stage bootloader tailored for one device. The second-stage bootloader is wired programmed in a masked ROM and which gets executed as the first binary after the device startup. It queries the SRAM PUF, deriving the device-dependent key $K$. This key is used to decrypt the third-stage bootloader, stored on non-volatile memory (e.g. flash memory). After successful decryption the third-stage bootloader will derive a second key $K'$. This key is subsequently used to decrypt the compressed kernel file of the firmware, which is also stored in NVM. This design assures that only a third-stage bootloader, which is encrypted by the correct, device-depended cryptographic key $K$, can be executed. 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 will only boot properly, if the correct combination of hardware and software is in place. The overall architecture is depicted in Figure 1.

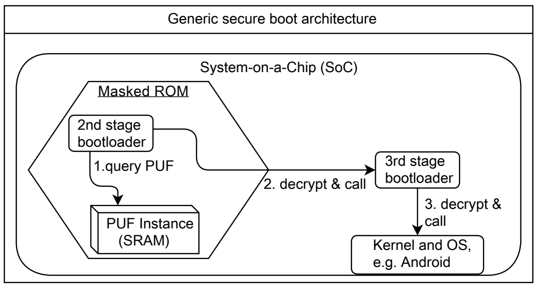

**Figure 1: Scheme of the reconstruction process of the ephemeral key inside the secure boot architecture.**

## 3.2 Attacker Model

We consider an attacker model where the attacker has digital access but can not physically attack the platform, which is the conventional model for TPM designs. The attacker represents malicious software, which tries to modify the kernel or the firmware of the embedded device. The intention of the malware is to achieve a stealthy and persistent installation of itself on the device for the purpose of logging PINs and secrets or mounting a hidden remote backdoor to the device. The malware tries to hook itself inside the system at kernel space level.

Additionally, the attacker model considers software, which can not be classified as malware. However, it is used to exploit vulnerabilities in the firmware to escalate the owner's privileges on the device to enable functions or install software, which are prohibited by the manufacturer's business model or terms and conditions. In the field of smartphones this process is referred to as 'rooting'.

## 4. PROOF OF CONCEPT

### 4.1 Hardware Platform

We evaluated our approach through an implementation on a System-on-a-Chip (SoC) platform. We selected a SoC architecture as most of the modern smartphones and tables are designed as such and the increasing usage of SoCs in mobile devices makes it likely that future devices will be designed using the same principle. We selected the PandaBoard [1] as basis for our implementation. The PandaBoard (ES) is an OMAP4430 (4460) based platform comprising of two ARM Cortex-A9 as well as two Cortex-M3 for signal processing. On basis of this hardware configuration we consider this to be an adequate reference setup since many of current smartphones show a similar ARM-based configuration. Next to the external 2 GiB DDR memory it consists of several on-chip memory (OCM) instances. The following OCM instances can be found on the platform: 1.) OCM Save-and-Restore ROM (4 KiB) 2.) OCM Save-and-Restore RAM (8KiB) 3.) Level-3 RAM (56 KiB).

Analysis of the OCM instances revealed, that only some portions of the RAM shows PUF-like behavior after startup. In particular the Level-3 OCM RAM can be partially used to extract a fingerprint for a given PandaBoard. The Level 3 OCM RAM (L3 RAM) consists of 56 KiB volatile on-Chip RAM. It is shared among different sub-modules of the PandaBoard, including the Cortex-M3 subsystem, the digital signal processing subsystem as well as IVA-HD, which is used for image and video hardware processing. The partitioning of the L3 RAM is defined by the L3 firewall logic. To extract a fingerprint from the startup values of the L3 OCM RAM, the values need to be extracted before any initialization is done. This is due to the fact that during the initialization the values residing in the L3 OCM RAM and consequently any identifying features of the PUF are overwritten. Thus, the L3 RAM was first evaluated whether it is not exposed to any initialization routine.

Figure 2 shows the bitmap of the whole L3 RAM from a PandaBoard shortly after the device gets out of reset.

The middle as well as high address space of the memory region is dominated by repeating structures. Since no other submodules are initialized at this early phase of boot process, we assume that these patterns represent structures used by the first-stage ROM code, which is shipped with every PandaBoard and is copied to OCM RAM even before the second-stage bootloader is called. The repeating structures might be caused by the ROM code's API interfaces. Furthermore, we assume that the OCM RAM is used in a similar way as a stack, since only higher addresses exhibit such patterns. In the area of the first 12 KiB (`0x40300000 - 0x40303000`) an apparently random distribution of zeros and ones can be seen.

This memory range refers to the part of the OCM RAM

[1]The first-stage bootloader usually resides in on-chip ROM and is pre-installed by the board manufacturer. It performs basic initialization like multi-core startup, clock configuration and power management and can not be modified by the IP vendor.

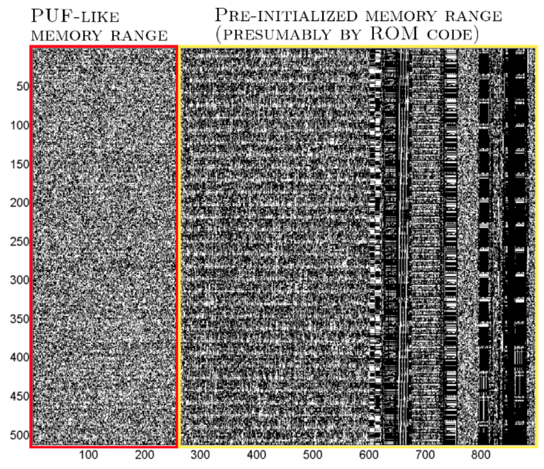

**Figure 2: Exemplary bitmap of a single PUF enrollment of a PandaBoard. The red area is used for fingerprint extraction. The yellow area contains initialized values and does not show PUF characteristics.**

we selected for further analysis. The results indicate that this portion of uninitialized memory shows good PUF characteristics (see Chapter 5.1) so that the modification of the bootloader – which includes the implementation of the Fuzzy Extractor and the residual architecture – could proceed.

## 4.2 Enrollment

During the enrollment phase the keys $K$ and $K'$, as well as the Helper Data $W$ are derived from a randomly chosen secret and a reference measurement of the SRAM. The keys are used to encrypt the 3rd-stage bootloader (`u-boot.img`), respectively the kernel image file (`uImage`). The Helper Data is required to reconstruct the keys from additional noisy measurements employing a Fuzzy Extractor. Eventually, the Helper Data, as well as the encrypted files are stored in non-volatile memory (e.g. the flash card). The enrollment procedure is performed by the manufacturer and is conducted once per device. Figure 3 depicts the enrollment procedure.

## 4.3 Reconstruction

In this project we use u-boot [7][5], which is one of the most widely used bootloaders. It integrates a second-stage bootloader (Memory Locator — `MLO`), which is small enough to fit into internal memory, and a third-stage bootloader (`u-boot.img`). The `MLO` performs some hardware initialization as well as the setup of external memory. Afterwards it calls the `u-boot.img`, which is copied to external memory, and initializes further hardware components and eventually calls the operating system kernel.

Our implementation uses the `MLO`, which is the first piece of code to be executed and which performs the extraction of the memory chip's fingerprint. This is done by reading the uninitialized bytes of the SRAM region that exhibit PUF characteristics, and processing it using the Fuzzy Extractor to derive the device-dependent key $K$. During the reconstruction process a noisy SRAM measurement $R'$ as well as

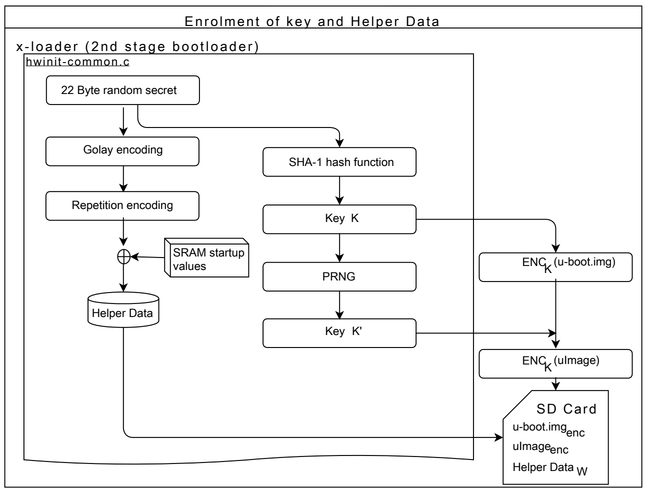

**Figure 3: Scheme of the enrollment process to generate the keys and Helper Data.**

$W$ is processed to reconstruct $K$, which subsequently decrypts `u-boot.img`. Accordingly, $K$ in combination with `u-boot.img` is used to derive $K'$, which is used to decrypt `uImage` and to continue the boot process. A detailed scheme of the reconstruction process is depicted in Figure 5.

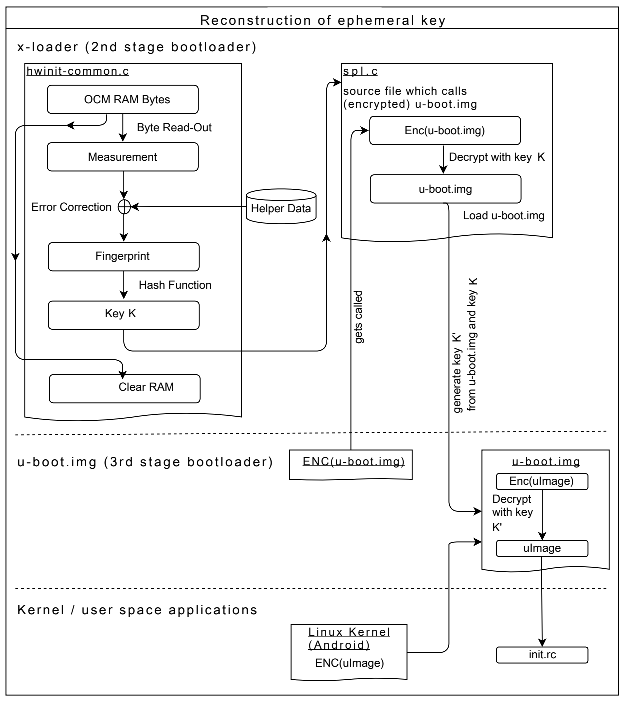

**Figure 4: Scheme of the reconstruction process of the ephemeral key inside the secure boot architecture.**

## 4.4 Fuzzy Extractor Design

To reproduce the secret key $K$ from various noisy measurements a Fuzzy Extractor is required. Following the suggestions of [3], we decided to implement a concatenated code comprising of a Golay code in combination with a linear repetition code. Furthermore, this concatenated code suffices to correct the amount of noise, which was estimated during the statistical analysis in Chapter 5.1. Using a binary Golay-(23,12,7) code in combination with a Repetition code with $r = 15$ repetitions, we are able to achieve a False Rejection Rate of almost $10^{-8}$ for key reconstruction given a maximum noise of 15%, which can be regarded as a reference value for SRAM PUF noise in literature.

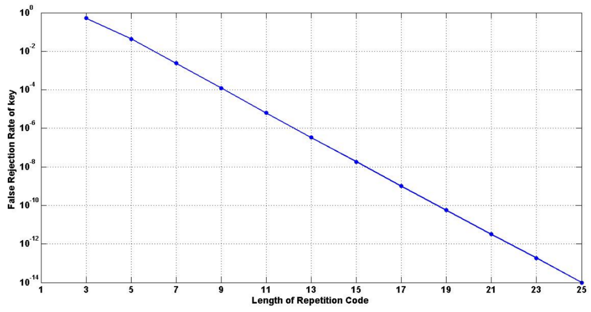

**Figure 5: False Rejection Rate of the key reconstruction using the Fuzzy Extractor.**

The implemented Fuzzy Extractor requires 900 bytes of SRAM data to reconstruct a 22 Byte secret. This secret is further processed by a SHA-1 hashing function to produce a key $K$. The implementation requires only 0,07% of the available SRAM memory.

## 5. EVALUATION

In this section we evaluate the PUF characteristics of intrinsic SRAM cells on the PandaBoard. The following measurements were performed on 5 PandaBoard instances. The devices include two versions of the platform - an early version of the PandaBoard equipped with an OMAP4430 and an advanced version, PandaBoard ES, based on an OMAP4460. The devices were triggered using a controller to repetitively turn the devices on, query the intrinsic PUF instance and turn it off. In between these queries a break of 15 seconds was introduced to give the SRAM the chance to uncharge. Using this setup, we conducted 1000 measurements per board. Figure 6 shows the hardware setup to automatically the PUF inside each of the PandaBoards.

### 5.1 Intrinsic PUF Characteristics

The explored intrinsic PUF instance was statistically analyzed regarding its quality to be used as an identifier for a given platform.

*Hamming Weight.* The Hamming Weight $HW(x)$ of individual measurements from the same PandaBoard indicates whether the start-up values are biased to either 0 or 1. This measurement gives a first impression on the randomness present in the start-up values. Our measurements show that the SRAM start-up values have $HW(x) = 48.53\%$ in the worst case, which is close to the ideal value with $HW(x) = 50\%$ representing almost the same amount of zeros and ones (Figure 7).

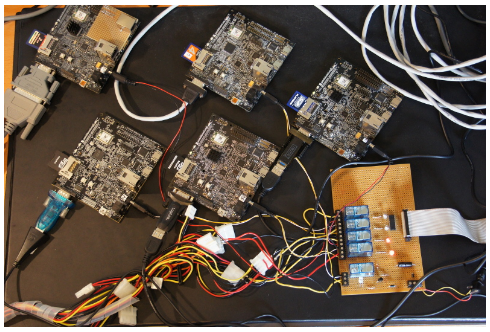

**Figure 6: Hardware setup for automated PUF measurements.**

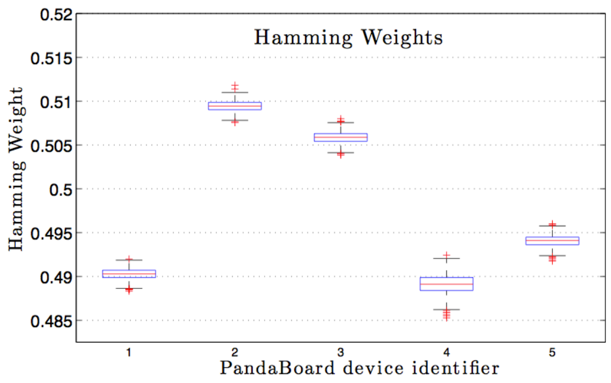

**Figure 7: Fractional Hamming Weights of SRAM start-up values.**

*Within-class Hamming Distance.* The within-class Hamming distance test gives an indication whether the PUF results are stable when queried repeatedly. This robustness of the start-up values is required to reliably identify a given device and subsequently reconstruct the corresponding cryptographic key. Therefore, an optimal value for the Within-class Hamming Distance is close to zero. However, all start-up values show a certain amount of instability (noise). The numbers in Figure 8 show a maximum within-class Hamming Distance of 4.67%.

*Between-class Hamming Distance.* The between-class Hamming distance test expresses whether the start-up values of different devices for the same challenge are independent. This measure states whether the start-up values can be used to uniquely identify a given device without enabling adversaries to predict a measurement for a second device on the basis of a given device for which the start-up values are already known. The optimal value for between-class Hamming Distance is 0.5, which refers to two start-up values with maximum difference. The statistics in Figure 9 attest almost optimal values with a minimum between-class Hamming Distance of 49.66%.

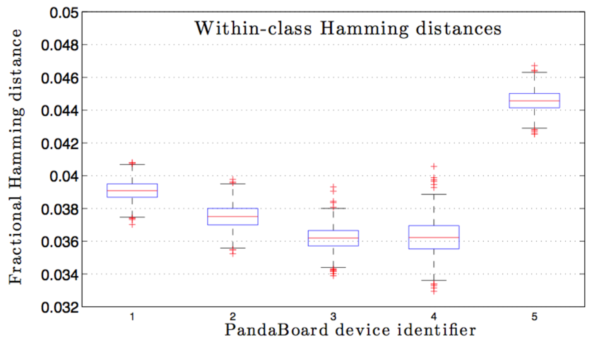

**Figure 8: Within-class fractional Hamming distance of SRAM start-up values.**

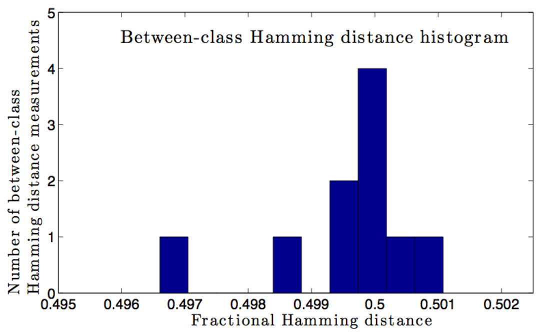

**Figure 9: Between-class fractional Hamming distance histogram.**

## 6. CONCLUSIONS

In this paper we proposed a light-weight secure boot approach using Physically Unclonable Functions found in commodity hardware. Our approach does not require additional hardware and sets up a chain-of-trust using standard computing components. The proposed solution guarantees a trusted state at boot time, which mitigates malicious modifications of the firmware or the operating system. We demonstrated the feasibility of our approach by implementing it on a state-of-the-art OMAP-based system-on-a-chip platform. The analysis of the extracted on-board PUF instance attested almost optimal performance.

## 7. REFERENCES

[1] PandaBoard Platform. `http://pandaboard.org/content/platform`. Last accessed on July 6th 2013.

[2] ARM. Building a Secure System using TrustZone Technology. Technical report, ARM, 2009.

[3] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls. Efficient Helper Data Key Extractor on FPGAs. In *Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '08, pages 181–197, 2008.

[4] P. Choi and D. K. Kim. Design of security enhanced TPM chip against invasive physical attacks. In *ISCAS*, pages 1787–1790, 2012.

[5] W. Denk. Das U-Boot – the Universal Boot Loader. `http://www.denx.de/wiki/U-Boot`. Last accessed on July 9th 2013.

[6] K. Dietrich and J. Winter. Secure Boot Revisited. In *Proceedings of the 9th International Conference for Young Computer Scientists, ICYCS 2008, Zhang Jia Jie, Hunan, China, November 18-21, 2008*, pages 2360–2365. IEEE Computer Society, 2008.

[7] X. Ding, Y. Liao, J. Fu, H. Huang, and W. Liu. Analysis of Bootloader and Transplantation of U-Boot Based on S5PC100 Processor. In *Proceedings of the 2011 Third International Conference on Intelligent Human-Machine Systems and Cybernetics - Volume 01*, IHMSC '11, pages 61–64. IEEE Computer Society, 2011.

[8] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 523–540. Springer Berlin Heidelberg, 2004.

[9] F-Secure. Bluetooth-Worm:SymbOS/Cabir. Technical report, F-Secure, 2004.

[10] D. E. Holcomb, W. P. Burleson, and K. Fu. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Trans. Comput.*, 58(9):1198–1210, Sept. 2009.

[11] G. F. Jerome Azema. M-Shield Mobile Security Technology: making wireless secure. Technical report, Texas Instruments, 2008.

[12] V. Leest, E. Sluis, G.-J. Schrijen, P. Tuyls, and H. Handschuh. Efficient Implementation of True Random Number Generator Based on SRAM PUFs. In D. Naccache, editor, *Cryptography and Security: From Theory to Applications*, volume 6805 of *Lecture Notes in Computer Science*, pages 300–318. Springer Berlin Heidelberg, 2012.

[13] R. Maes, P. Tuyls, and I. Verbauwhede. Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 332–347. Springer-Verlag, 2009.

[14] R. Maes and I. Verbauwhede. Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions. In A.-R. Sadeghi and D. Naccache, editors, *Towards Hardware-Intrinsic Security*, Information Security and Cryptography, pages 3–37. Springer Berlin Heidelberg, 2010.

[15] D. Maslennikov. Mobile Malware Evolution: Part 6. Technical report, Kaspersky Labs, 2013.

[16] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*. The Internet Society, 2011.

[17] TrendLabs 3Q 2012 Security Roundup. Android Under Siege: Popularity Comes at a Price. Technical report, TrendLabs, October 2012.

[18] D.-H. You and B.-N. Noh. Android platform based linux kernel rootkit. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, pages 79–87, Washington, DC, USA, 2011. IEEE Computer Society.

## A.2 Paper: "Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers"

**Authors:** Anthony Van Herrewege, Vincent van der Leest, André Schaller, Stefan Katzenbeisser, Ingrid Verbauwhede
**Venue:** International Workshop on Trustworthy Embedded Devices (TrustED) 2013
**Date:** November 4th - 8th, 2013

**Status:** Will appear at the International Workshop on Trustworthy Embedded Devices. An extended version of the paper (which forms the appendix), can be found on the IACR ePrint archive (`http://eprint.iacr.org/2013/304.pdf`)

# Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers

Anthony Van Herrewege[1], Vincent van der Leest[2], André Schaller[3], Stefan Katzenbeisser[3], and Ingrid Verbauwhede[1]

[1] KU Leuven Dept. Electrical Engineering-ESAT/SCD-COSIC and iMinds
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
`firstname.lastname@esat.kuleuven.be`
[2] Intrinsic-ID, Eindhoven, The Netherlands
`http://www.intrinsic-id.com`
[3] Security Engineering Group, Technische Universität Darmstadt and CASED, Germany
`lastname@seceng.informatik.tu-darmstadt.de`

**Abstract.** The generation of high quality random numbers is crucial to many cryptographic applications, including cryptographic protocols, secret of keys, nonces or salts. Their values must contain enough randomness to be unpredictable to attackers. Pseudo-random number generators require initial data with high entropy as a seed to produce a large stream of high quality random data. Yet, despite the importance of randomness, proper high quality random number generation is often ignored. Primarily embedded devices often suffer from weak random number generators. In this work, we focus on identifying and evaluating SRAM in commercial off-the-shelf microcontrollers as an entropy source for PRNG seeding. We measure and evaluate the SRAM start-up patterns of two popular types of microcontrollers, a STMicroelectronics STM32F100R8 and a Microchip PIC16F1825. We also present an efficient software-only architecture for secure PRNG seeding. After analyzing over 1 000 000 measurements in total, we conclude that of these two devices, the PIC16F1825 cannot be used to securely seed a PRNG. The STM32F100R8, however, has the ability to generate very strong seeds from the noise in its SRAM start-up pattern. These seeds can then be used to ensure a PRNG generates high quality data.

**Keywords:** Secure seeds, PRNG, Static RAM, Security

## 1 Introduction

The generation of high quality random numbers is crucial to many cryptographic applications. Almost every cryptographic protocol involves the use of keys, nonces or salts which are unpredictable to attackers. Such values must exhibit a sufficient degree of randomness, i.e., contain enough entropy. In addition, re-keying is applied regularly to prevent a wear-out effect of secret keys. Finally, public-key cryptosystems, such as RSA and ElGamal, rely on random numbers to generate public/private key pairs.

Yet, despite its importance, proper high quality random number generation is often ignored. Random data with too little entropy results in weak keys, nonces or salts, which can then be guessed with minimal effort, thereby compromising even the strongest cryptosystem. Hence, the quality of random data ultimately affects the level of security of cryptographic primitives and protocols in practice. Although many cryptographically secure pseudo-random number generators (PRNG) exist, all of them require to be seeded with initial data containing sufficient entropy. Once seeded, they are able to generate high quality random output for long periods of time. Providing PRNGs with a low quality initial seed, however, will cause them to generate weak, predictable output.

Neglecting to ensure sufficient entropy in PRNG seeds gave rise to several security incidents. A famous case was the OpenSSL implementation in Debian [15]: by accidentally decreasing the number of available randomness sources for seeding, the generated random numbers became predictable. This incident affected numerous TLS/SSL connections, keys for SSH accounts, as well as the security of Tor users [4]. More recently, Heninger et al. [7] and Lenstra et al. [11] conducted an Internet-wide survey and looked for security problems in public keys and certificates of TLS and SSH servers, caused by low quality random number generation. The authors were able to recover private keys of several devices due to common factors in public RSA keys. Their results indicate that primarily embedded devices, such as routers, firewalls and VPN appliances, are affected. The source of these problems are likely PRNGs that were not seeded with high entropy data on start-up.

In this work, we focus on identifying and qualifying Static Random Access Memory (SRAM) in commercial off-the-shelf (COTS) microcontrollers as an entropy source for PRNG seeding. We take advantage of the fact that the start-up values of SRAM are noisy. This noise is collected upon boot time to derive a high quality, high entropy PRNG seed by applying a hash function to the initial memory contents. We measure and evaluate the entropy in SRAM start-up patterns in two common types of microcontrollers, an STMicroelectronics STM32F100R8 (ARM Cortex-M3) and a Microchip PIC16F1825. Furthermore, we suggest an architecture for seed extraction and pseudo-random number generation, which makes efficient use of the available resources in a COTS microcontroller.

The paper is structured as follows. After surveying related work in Section 2, we analyze and evaluate the noise in the SRAM start-up patterns of the aforementioned microcontrollers in Section 3. In Section 4, we present the architecture for an efficient SRAM-based secure seed generator and PRNG. Furthermore, we present our attacker model and discuss practical aspects relating to the implementation of our architecture. Finally, we conclude the paper in Section 5.

## 2 Related Work

*Random number generation.* In order to generate random numbers for cryptographic applications on microcontrollers, two basic methods can be used. The first method requires a physical source, which is truly random and from which bits can be derived directly. Such non-deterministic sources derive their randomness from underlying physical properties that exhibit unpredictable behaviour. Examples of such sources of randomness in chips are free running oscillators connected to a shift register [19] and noise on the lowest bits of AD converters [17], but many more exist. There are two important downsides to most of these physical RNG constructions. Firstly, they require specific hardware to extract the randomness from the physical entities on the device. Secondly, the throughput of such RNGs is generally relatively low. This is problematic when large streams of random bits are required for cryptographic applications.

The second approach to generate randomness is by using PRNGs, which are deterministic algorithms. An introduction to PRNGs can be found in [1]. The output of such a generator only seems random to observers without prior knowledge. However, if an observer knows which data has been used as a seed for the PRNG, then he will be able to calculate all output values of the generator. Hence, this seed value should be randomly chosen (and kept secret). The upside of this type of generator is that it can be implemented completely in software and therefore does not require any hardware additions to a microcontroller. Also, it can produce a stream of (pseudo-)random output bits at a high throughput rate.

The benefits of a PRNG greatly outweigh those of a true random number generator on a COTS microcontroller. The necessity of generating a truly random seed is of crucial importance though. Our goal is to identify and evaluate methods that can be used to generate strong seeds for a PRNG and that are already available in COTS microcontrollers, thus requiring no hardware modifications.

*SRAM as sources of entropy.* Our approach of generating a seed value is based on random noise extracted from the power-up state of SRAM modules, which are part of COTS microcontrollers. SRAM bit cells are designed as cross-coupled inverters, which exhibit a bi-stable behaviour. When powered on these cells eventually settle from a meta-stable state to a stable state, either zero or one. It was shown by Guajardo et al. [6] that memory cells are often biased to zero or one, due to uncontrollable physical conditions during the manufacturing stage leading to one of the inverters being slightly stronger than the other. Some cells, however, will be almost perfectly symmetric, which leads them to settle to an unpredictable value at start-up. It is the noise due to these cells which we exploit in order to generate high quality random seeds.

The general idea of using SRAM as a source for PRNG seeds was investigated in [8] as well as in [10]. However, the former paper proposes to use a universal hash to generate a single random number at start-up. This technique is then verified on an external SRAM module. However, it is not investigated whether the approach works on the embedded SRAM in COTS microcontrollers. In the latter paper, the feasibility of creating a strong PRNG with the use of random data from an ASIC containing SRAM-based Physically Unclonable Functions (PUFs) [12] is investigated. In contrast, our goal is to identify COTS microcontrollers which can be used without any hardware modifications to support high quality random number generation and hence cryptographic protocols.

In Mowery et al. [18], the authors gather entropy from the clock jitter between different clock domains on a CPU. Their approach is quite slow, however, and obviously does not work on embedded devices with only a single clock domain. In cases where their approach is feasible, it can be combined with our method in order to increase the amount of gathered entropy.

## 3 Evaluation of Entropy in SRAM Start-up Values

For the purpose of extracting a random seed from SRAM start-up values, it is important to investigate their entropy contents. In this section, our approach to quantify the entropy quality of the SRAM patterns (namely the calculation of min-entropy) is explained. We will also present the hard- and firmware used to measure the SRAM start-up pattern of two popular COTS microcontrollers. Finally, we show and discuss the measurements for these two devices under different ambient conditions.

The first investigated microcontroller is the 32-bit STM32F100R8 by STMicroelectronics, an ARM Cortex-M3 chip. The second one is the 8-bit PIC16F1825 by Microchip, part of Microchip's range of high-end 8-bit processors. Both of these chips were chosen for their popularity. The STMicroelectronics chip was chosen due to the Cortex-M family being ARM's fastest licensing processor family to date. The Cortex-M family was licensed 168 times by Q4 2012 [13], with 23 billion of these chips sold last year. Microchip has the 4th largest market share in the extremely fractioned microcontroller market [14].

### 3.1 Method of deriving min-entropy

To extract a high quality seed from the SRAM start-up values we have to examine their randomness properties in terms of entropy. In particular, the amount of entropy must be present in the noise of SRAM start-up patterns should be determined. For this purpose we will be calculating the min-entropy in the same manner as was done in [10]. This method is based on the NIST specification [3] that defines min-entropy as the worst-case (i.e., the greatest lower bound) measure of uncertainty for a random variable.

For a binary source, we can define the min-entropy as

$$H_{min} = -\log_2(\max(p_0, p_1)),$$

where $p_0$ and $p_1$ are the probabilities of 0 and 1 occurring. Assuming that all bits from the SRAM start-up pattern are independent, each bit $i$ can be viewed as an individual binary source. For each of these sources we estimate the probabilities $p_0^i$ and $p_1^i$ of powering up in state 0 or 1, by repeatedly measuring the power-up values of the SRAM. In case $m$ subsequent measurements are performed, $p_0^i$ denotes the number of occurrences of a zero, divided by $m$ and $p_1^i = 1 - p_0^i$. For $n$ independent sources (where $n$ is the length of the start-up pattern), we have:

$$H_{min} = \sum_{i=1}^{n} -\log_2(\max(p_0^i, p_1^i))).$$

Hence, under the assumption that all bits are independent, we can sum the entropy of each individual bit to derive the min-entropy of the entire SRAM. In the remainder of this work, we generally denote the available min-entropy as a percentage of the total available SRAM size.

### 3.2 Measurement setup

In this subsection, we present the soft- and hardware setup used to evaluate COTS microcontrollers. First, we present the functionality and requirements of the firmware that has to be put into each microcontroller to be measured. Thereafter, we describe the hardware construction used to extract start-up values for later evaluation.

*Firmware design* Every microcontroller to be measured should be programmed with firmware that, on power-up, initializes the serial port and then starts transmitting the value of each SRAM byte in sequence. Once finished, it should enter an idle loop. Care should be taken not to use any of the SRAM storage while doing this. Most microcontrollers have a several working registers to store variables, such as a pointer to the current SRAM byte, and thus this will be easy to achieve. However, some microcontrollers, such as the Microchip PIC16 family, only have a single working register and therefore, in order not to write data to any SRAM byte, some variables will have to be stored in unused configuration registers.

*Hardware setup.* To get some initial measurements of the SRAM power-up patterns, we first conduct our experiments manually. In this setup, the microcontroller to be measured has its power lines and serial port connected to an external serial TTL–to–USB converter. The converter is connected to a self-powered USB hub. After an SRAM measurement has been taken, power to the microcontroller is switched off (i.e. left floating) for at least 10 seconds. This is to ensure that the microcontroller has discharged completely and that the SRAM will contain fresh data on the next power-up. Although this discharging cycle works fine for the STM32F100R8 devices, it does not for the PIC16F1825 devices, which keep their SRAM values for over 10 minutes when their supply pins are left floating.

In order to extract start-up patterns faster and efficiently, we created a custom measurement board. The requirements for this board are:

1. Allow connection of many microcontrollers at once.
2. Be extensible with regards to number of attached microcontrollers.
3. Support remote setup.
4. Make automated, unsupervised measurements possible.
5. Support any realistic baud rate.
6. Support any arbitrary SRAM size.
7. Supply upwards-going, fast rising ($\leq$2 ms) Vcc signals.
8. Actively discharge microcontrollers that are not being measured.

Requirements 1 and 2 are satisfied by using (de)multiplexers for the power supply and serial transmission (TX) lines of the attached microcontrollers. The controller board interfaces with a PC, thereby meeting requirements 3 and 4. The controller clock signal is generated with a specialized clock, and the baud rate can also be set though the PC interface, thus fulfilling requirement 5. Requirement 6 is met by detecting when the TX line of the currently powered microcontroller goes idle, at which point the controller board advances to the next connected microcontroller. In order to generate realistic start-up patterns, requirement 7 should be met. We used an oscilloscope to verify that this was the case for our controller board. Finally, requirement 8 is necessary in order to erase the state of the SRAM completely on power-down. The demultiplexer on our controller board connects non-active power lines to ground, thereby this last requirement is met as well.

A simplified schematic of our design is shown in Fig. 1. In its current state, it allows us to connect up to 16 microcontrollers. This can be extended to at least 1024 devices, in case this should prove necessary.

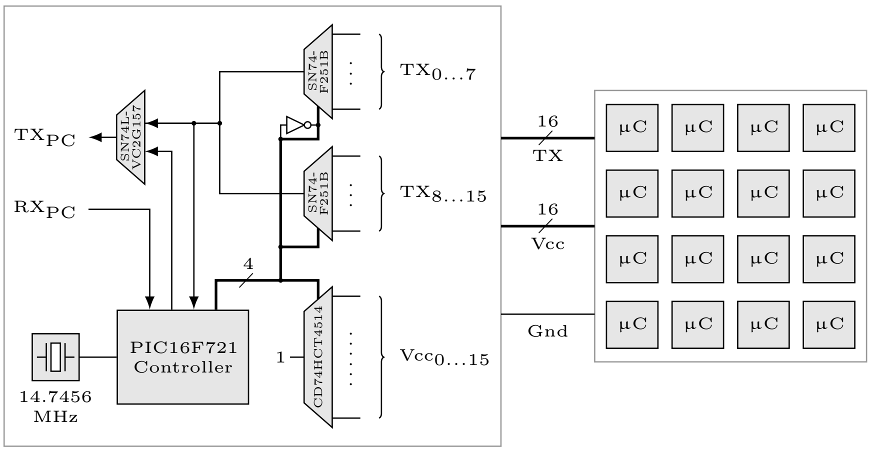

**Fig. 1.** High-level schematic of the measurement controller board (*left*) with a board of microcontrollers to be measured attached (*right*).

### 3.3 STMicroelectronics STM32F100R8

The first device that has been tested for entropy in the SRAM start-up values is the STM32F100R8 from STMicroelectronics. This is a 32-bit ARM Cortex-M3 device with 8 KiB of SRAM. Of this device 10 samples have been used to perform a large number of measurements. An example of a start-up pattern (measured at +25 °C) of the SRAM in this microcontroller can be found in Fig. 2.

To make sure that the STM32F100R8 provides sufficient noise entropy under different circumstances, measurements have been performed at −30°C, +25°C and +90°C. Using these measurements the min-entropy has been derived (with the method described in Section 3.1), the results[4] for the different conditions can be found in Table 1.

**Table 1.** Min-entropy results for STM32F100R8 SRAMs at different temperatures. Min-entropy denoted as percentage of total available SRAM.

| Temp. | Microcontroller ID | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| [°C] | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| −30 | 5.3% | 5.3% | 5.4% | 5.5% | 5.4% | 5.3% | 5.4% | 5.2% | 5.3% | 5.8% |
| +25 | 6.6% | 6.6% | 6.7% | 6.8% | 6.7% | 6.5% | 6.8% | 6.5% | 6.7% | 6.7% |
| +90 | 6.3% | 6.5% | 6.5% | 6.6% | 6.2% | 6.5% | 6.5% | 6.2% | 6.5% | 6.5% |

From the results in Table 1 it is clear that the STM32F100R8 devices contain a minimum amount of 5.2% min-entropy under all tested circumstances. Given the

[4] Min-entropy is expressed here as a percentage of the length of the start-up pattern. Hence, 6.0% in this 8 KiB memory is approximately equal to 491.5 bytes min-entropy.

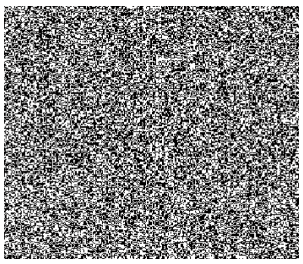

**Fig. 2.** Example start-up pattern of STM32F100R8 (8 KiB SRAM) at +25 °C. White represents a bit with value 0, black a bit with value 1.

fact that the measured SRAMs have a size of 8KiB, it is evident that by using these memories as input for a hash function it is no problem to derive a truly random seed for a PRNG[5]. If for example, assuming the entropy is evenly spread out over the entire SRAM, we would like to derive a truly random seed of 256 bits and consider a min-entropy of 3% (which is on the safe side, given the lowest min-entropy of 5.2% from the analysis), the required amount of SRAM to derive this seed is only 1.04KiB. For a more cautious approach, in which no assumptions are made about the entropy distribution, see Appendix B.

### 3.4 Microchip PIC16F1825

The second commercially available microcontroller that has been evaluated, is the Microchip PIC16F1825. This is an 8-bit microcontroller with 1 KiB of SRAM. Under the same conditions as described in the previous section, a large number of measurements of the SRAM start-up patterns of 16 different devices have been performed. A plot of one of these start-up patterns (measured at +25 °C) is given in Fig. 3. It is evident from this plot that there is severe biasing in the start-up pattern. The plot clearly shows that the bits from the PIC16F1825 memories possess a pattern which is far from random. To be more precise: the bits of every alternating byte have a preference to start-up either as a 0 or a 1. A pattern as can be seen in Fig. 3 is present in every PIC16F1825 device measured. The preference towards 0 or 1 for each byte results in a lower noise entropy, because it is less common for these bits to flip since they have a preferred state to start in. Using these measurements, the min-entropy of the SRAM noise has been determined in the same way as for the STM32F100R8 devices. The resulting min-entropies at different temperatures can be found in Table 2.

In comparison to the results of the STM32F100R8 devices, it is clear that the noise entropy for the PIC16F1825 is significantly lower. For the measurements at room and high temperatures this can be explained by the severe biasing of the start-up pattern, which has been discussed already. For the low temperature (at which the min-entropy is

[5] A less extensive evaluation STM32F051R8 and STM32F100RB devices seems to suggest that other devices in the STM32 family contain an equally high amount of entropy in their SRAM start-up patterns.

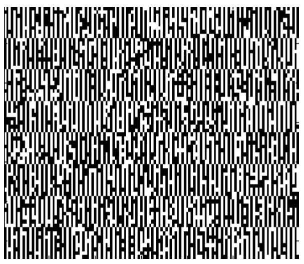

**Fig. 3.** Example start-up pattern of PIC16F1825 (1 KiB SRAM) at +25 °C. White represents a bit with value 0, black a bit with value 1.

**Table 2.** Min-entropy results for PIC16F1825 SRAMs at different temperatures. Min-entropy denoted as percentage of total available SRAM.

| Temp. | Microcontroller ID | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| [°C] | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| −30 | 0.9% | 1.0% | 0.3% | 0.2% | 0.5% | 0.2% | 0.2% | 0.2% |
| +25 | 1.9% | 2.0% | 1.8% | 1.8% | 1.9% | 1.9% | 1.8% | 2.0% |
| +90 | 3.2% | 3.2% | 3.2% | 3.8% | 3.3% | 3.5% | 3.7% | 3.5% |
| [°C] | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| −30 | 0.1% | 0.3% | 0.1% | 0.2% | 0.2% | 0.8% | 1.7% | 0.1% |
| +25 | 1.7% | 1.7% | 1.8% | 2.1% | 1.8% | 1.7% | 1.6% | 1.7% |
| +90 | 3.6% | 3.6% | 3.8% | 4.1% | 3.3% | 3.5% | 4.0% | 3.7% |

very close to 0 for most of the devices), the reason is different. Our observation is that at these temperatures the SRAM start-up patterns exhibit a significant decrease in the Hamming weight for all tested devices. For all devices the Hamming weight was very close to 0, which means that almost all bits of the memory start-up as a 0 and only very few (in the order of magnitude of 1%) as a 1. These results clearly show that by exposing the PIC16F1825 to (extremely) low temperatures it is possible to make the start-up pattern of the SRAM more predictable. We shall call this controlled decrease of entropy a "freezing attack". Such an attack scenario is outside the scope of our attacker model (see Section 4.1), since it requires for an attacker to have physical access to the device to freeze the memory. However, this phenomenon does present a major issue for usability of the PIC16F1825, because it will not be possible to generate sufficient entropy for the PRNG seed when ambient temperatures are sufficiently low (e.g. during wintertime in large parts of the world).

Based on the problems detected at low temperatures, the clearly visible patterns within the SRAM start-up values (see Fig. 3), and the very small security margin hinted at by the min-entropy calculations, we advise strongly against using PIC16F1825 devices to generate a secure seed for PRNG initialization[6].

### 3.5 Discussion of measurement results

From the measurement results in the two previous sections it becomes clear that the two investigated device types behave very differently. The STM32F100R8 devices show great results when it comes to deriving entropy from the noise on SRAM start-up patterns, while the PIC16F1825 are clearly unfit for the purpose of extracting a truly random seed from this noise. Besides the simple conclusion that when one wants to implement a PRNG on a microcontroller, which uses a truly random seed derived from SRAM start-up noise, one should not use the PIC16F1825 but rather the STM32F100R8 devices, this section also takes a closer look at trends that become apparent from the results of these two devices.

*Dependencies on ambient temperature.* Fig. 4 provides a visual representation of the results of the min-entropy measurements from the STM32F100R8 and PIC16F1825 chips at different temperatures from Table 1 and Table 2. In this plot, measurements at the same temperature are encoded using the same shape for data points.

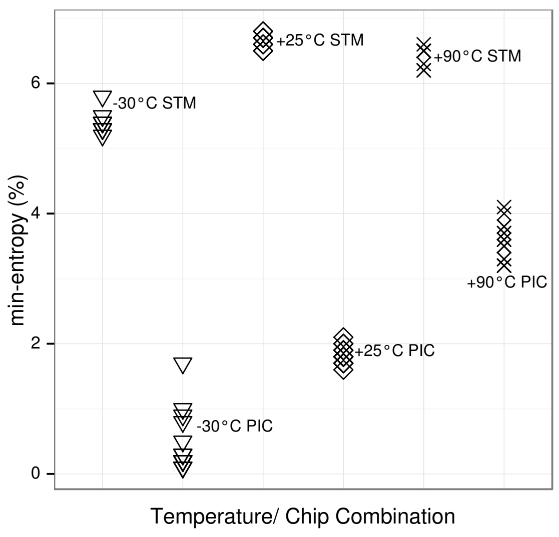

**Fig. 4.** Scatter plot of min-entropy at various temperatures for STM32F100R8 and PIC16F1825.

[6] A less extensive evaluation of PIC16F877A and PIC16F721 devices seems to suggest that other devices in the PIC16F family have similarly low entropy in their SRAM start-up patterns.

From Fig. 4 we can derive two clear trends. The first is, as already concluded before, the fact that the min-entropy of the STM32F100R8 memories is greater than that of the PIC16F1825s under any tested circumstance. More interesting is the second trend, which shows that the behavior over different temperatures for these two devices is very different. While the min-entropy of the STM32F100R8 is reasonably stable over temperature, the min-entropy of the PIC16F1825 shows a clear (perhaps even linear) correlation with the temperature. In other words: the colder the ambient temperature, the lower the min-entropy of these start-up patterns.

This behaviour again shows that the PIC16F1825 devices are very much unsuitable for use in the proposed random number generator. An uncontrollable variable, such as the ambient temperature, should never be able to influence the amount of entropy that will be available in the seed of a PRNG.

*Dependencies on start-up power curve.* During our experiments we have made an attempt to increase the min-entropy in the start-up patterns of the PIC16F1825 devices (at room temperature) by making alterations to their power circuitry. Since altering circuitry goes against the principle employed in this paper, i.e. using unmodified COTS devices, we will not consider these results in the main story of the paper. More details on this power-up curve dependency can be found in Appendix A. However, we would like to point out that altering the shape of the power-up curve on the supply pins of the PIC16F1825 devices has resulted in a reduction of the bias in the SRAM start-up patterns, which increases the entropy of these memories. This observation shows the considerable possibility that the biasing in the start-up pattern is caused by internal circuitry that is in charge of supplying power to the SRAM. It is possible that (analog) components inside the PIC16F1825 distort the supply curve before it is able to power-up the SRAM. Unfortunately Microchip does not provide information about their silicon implementation, which makes it impossible for us to verify what is happening inside the devices.

## 4 Architecture of an SRAM-based RNG

SRAM start-up values, as analyzed in the previous section, can be used to derive PRNG seeds in an efficient and lightweight manner on low-cost COTS microcontrollers without the need for extra hardware (along the lines of [10]). In a nutshell, we measure the start-up contents of SRAM cells right after power up and post-process them in order to extract a seed (see Fig. 5):

1. First, the device is powered up. Care should be taken that the power-up voltage follows a nice curve, as explained in Appendix A. Furthermore, the device should be properly discharged before power-up, such that the SRAM is completely cleared.
2. In the second step, the seed generation algorithm is run: the code reads the entire SRAM content and applies a hash function to it to derive the seed, as suggested in [3, 5, 9]. This step ensures a consistently high entropy in the seed value. Note that this algorithm must be the first code that executes on the device in order to ensure that the SRAM contains uninitialized data.

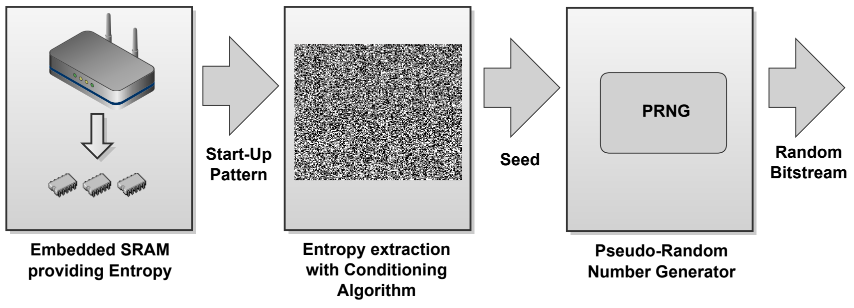
Start-Up Pattern
Seed
PRNG
Random Bitstream
Embedded SRAM providing Entropy
Entropy extraction with Conditioning Algorithm
Pseudo-Random Number Generator

**Fig. 5.** Secure PRNG seeding based on noise in SRAM start-up pattern.

Care should be taken when implementing the hash function: first of all, the hash function must be designed to be lightweight, as the target embedded platform probably has limited storage and computational power. Furthermore, any temporary storage used by the hash function will overwrite parts of the SRAM start-up patterns, which need to be excluded from seed derivation (in the worst case 8 bits of entropy from the SRAM start-up pattern are lost for each byte used in the hash function implementation). Therefore, it is important to select a hash function that requires a small program size and has limited memory consumption; Appendix B discusses suitable implementations of the hash function.

3. Finally, the generated hash is used to seed a PRNG, which can then be used to obtain a stream of random numbers. Implementations for PRNGs are extensively documented in the literature (for example, see [3, 5, 9]). For use in low-cost devices we suggest to apply a block cipher in counter (CTR) or output feedback (OFB) mode, which are known to before as cryptographically secure PRNGs. The reason for this choice is that a block cipher implementation is most likely already available in a device which requires cryptographic algorithms; this reduces both implementation costs and code size compared to implementing a dedicated PRNG algorithm. With the appropriate construction, a block cipher can also be used as a hash function, further decreasing costs and code size (see Appendix B).

### 4.1 Security considerations and attacker model

Crucial for security is to maintain the unpredictability of the data stream produced by the PRNG. Once an attacker knows the seed, the entire stream becomes predictable. Thus, care needs to be taken that no other algorithm has access to the seed value — approaches to achieve this are the subject of a separate field of embedded cryptography research and thus outside of the scope of this paper.

In this work we assume an attack scenario in which an adversary has no direct physical access to the microcontroller. Otherwise it would be impossible to ensure that the power-up SRAM value remains secret, since an adversary can use a debugging interface such as JTAG to halt the microcontroller during start-up, read out the data and then let the start-up process continue.

To limit the exposure of the initial SRAM state and prevent attacks where the seed is re-calculated from SRAM content, all SRAM (except for the seed value) should be cleared immediately after seed generation. This can be achieved by making sure that the seeding algorithm is the very first code that runs on power-up and that the algorithm is executed atomically. Methods to ensure this, such as disabling interrupts and preventing unauthorized firmware modifications, are outside the scope of this paper.

Finally, in order to guarantee proper SRAM resets in between power-cycles of the microcontroller, care should be taken that the microcontroller's positive supply lines are grounded when the device shuts down. If this is not done, it might power up with old, predictable data with low entropy still present in SRAM.

## 5 Conclusion

In this work, the problem of weak seeds used to initialize PRNGs was addressed. Such weak seeds lead to the PRNG generating predictable random numbers. We presented a lightweight software-only approach to generate secure seeds on commercial off-the-shelf microcontrollers. The source of entropy used to generate these seeds is the noise present in SRAM at start-up. In order to support that such an approach is feasible with COTS microcontrollers, we measured and evaluated this noise at various ambient temperatures in two popular devices, the STMicroelectronics STM32F100R8 (an ARM Cortex-M3) and the Microchip PIC16F1825. Our analysis shows that the SRAM start-up patterns of the PIC16F1825 devices contain very little entropy, which are thus unfit for secure seed generation. Furthermore, we address the peculiarities of these devices under both temperature and supply voltage variations. The SRAM start-up patterns of the STM32F100R8 devices on the other hand contain a large amount of entropy, thereby showing that our approach is indeed feasible and that unmodified COTS microcontrollers using a software-only approach can generate secure seeds for PRNGs.

## Acknowledgements

This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007) and by the Flemish iMinds projects. In addition, this work is supported in part by the Hercules Foundation AKUL/11/19, and by the European Commission through the ICT programme under contract FP7-ICT-2011-284833 PUFFIN.

## Bibliography

[1] Babaei, M., Farhadi, M.: Introduction to Secure PRNGs. International Journal of Computer Network and Security 4(10), 616–621 (2011)

[2] Balasch, J., Ege, B., Eisenbarth, T., Gérard, B., Gong, Z., Güneysu, T., Heyse, S., Kerckhof, S., Koeune, F., Plos, T., Pöppelmann, T., Regazzoni, F., Standaert, F.X., Assche, G.V., Keer, R.V., van Oldeneel tot Oldenzeel, L., von Maurich, I.: Compact Implementation

and Performance Evaluation of Hash Functions in ATtiny Devices. IACR Cryptology ePrint Archive 2012, 507 (2012)

[3] Barker, E., Kelsey, J.: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. NIST Special Publication 800-90A (Jan 2012)

[4] Dingledine, R.: Tor Security Advisory: Debian Flaw Causes Weak Identity Keys (May 2008), https://lists.torproject.org/pipermail/tor-announce/2008-May/000069.html

[5] Eastlake, D., Schiller, J., Crocker, S.: Randomness Requirements for Security. RFC 4086 (Best Current Practice) (Jun 2005), http://www.ietf.org/rfc/rfc4086.txt

[6] Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: FPGA Intrinsic PUFs and Their Use for IP Protection. In: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems. Lecture Notes in Computer Science, vol. 4727, pp. 63–80 (Sep 2007)

[7] Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In: Proceedings of the 21st USENIX Security Symposium (Aug 2012)

[8] Holcomb, D.E., Burleson, W.P., Fu, K.: Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. IEEE Trans. Comput. 58(9), 1198–1210 (Sep 2009)

[9] Kelsey, J., Schneier, B., Ferguson, N.: Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In: Proceedings of the 6th Annual International Workshop on Selected Areas in Cryptography. pp. 13–33. SAC '99 (1999)

[10] van der Leest, V., van der Sluis, E., Schrijen, G.J., Tuyls, P., Handschuh, H.: Efficient Implementation of True Random Number Generator Based on SRAM PUFs. In: Naccache, D. (ed.) Cryptography and Security. Lecture Notes in Computer Science, vol. 6805, pp. 300–318. Springer (2012)

[11] Lenstra, A.K., Hughes, J.P., Augier, M., Bos, J.W., Kleinjung, T., Wachter, C.: Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064 (2012)

[12] Maes, R.: Physically Unclonable Functions: Constructions, Properties and Applications. Ph.D. thesis, KU Leuven (2012), Ingrid Verbauwhede (promotor)

[13] ARM Holdings PLC: Results for the Fourth Quarter and Full Year 2012 (Feb 2013), http://ir.arm.com

[14] Databeans Inc.: Microcontroller Market Share: In 3 Dimensions (Mar 2012)

[15] Debian Security: DSA-1571-1 OpenSSL – Predictable Random Number Generator. Tech. rep. (May 2008), http://www.debian.org/security/2008/dsa-1571.en.html

[16] Information Technology Laboratory NIST: FIPS 140-3: Security Requirements for Cryptographic Modules, Annex A: Approved Security Functions for FIPS PUB 140-3 (Jul 2009), draft

[17] Moro, T., Saitoh, Y., Hori, J., Kiryu, T.: Generation of Physical Random Number Using the Lowest Bit of an A-D Converter. Electronics and Communications in Japan (Part III: Fundamental Electronic Science) 89(6), 13–21 (2006)

[18] Mowery, K., Wei, M., Kohlbrenner, D., Shacham, H., Swanson, S.: Welcome to the Entropics: Boot-Time Entropy in Embedded Devices. In: IEEE Symposium on Security and Privacy (May 2013), to be published

[19] Petrie, C., Connelly, J.: A Noise-based IC Random Number Generator for Applications in Cryptography. Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on 47(5), 615 –621 (May 2000)

## A Dependencies of SRAM Entropy on Start-up Power Curve

In Section 3.4, we have seen that the SRAM start-up patterns of the tested PIC16F1825 devices showed severe biasing at room temperature. Based on the authors' practical experience, the most common reason for biased start-up patterns (besides those due to low temperatures, which were already shown in the same section) is a due to the suppy voltage ramp-up curve on the SRAM cells. Measuring the supply voltage curve applied to the PIC16F1825 by our controller board shows a ramp as can be seen on the left side of Fig. 6. Unfortunately, this is a ramp-up curve which is very common for powering SRAMs and usually results in proper (unbiased) start-up patterns. However, this does not mean that the supply voltage curve on the actual SRAMs cannot be the root-cause. We can only measure the curve on the external pins of the devices and we do not know what happens internally with the supply voltage before it reaches the SRAM. It is possible that there are (analog) components connected to the power supply, which distort the ramp-up on the SRAM. Unfortunately Microchip does not provide information about their silicon implementation, which makes it impossible for us to verify what is happening inside the devices.

In an attempt to make the start-up pattern of the SRAMs more random, we have performed experiments with varying the shape of the supply voltage curve on the supply pins. One of the shapes that we have tried (and the only one that improved our results) consists of a short pulse on the supply pin just before the actual ramp-up curve. An example of such a shape can be seen on the right side of Fig. 6. In this shape the height and width of the pulse can be varied.

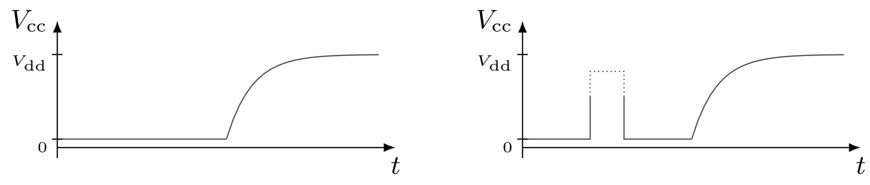

**Fig. 6.** Original supply voltage curve (*left*, normal for SRAMs) and altered curve (*right*).

With the "new" supply voltage curve on the supply pins of the PIC16F1825s, some devices presented start-up patterns which turned out to be more random than the original patterns. An example of such a new pattern can be found in Fig. 7. Unfortunately, we were not able to make the start-up patterns of all PIC16F1825 devices more random with this method. Also, the height of the pulse before the voltage ramp-up curve (which resulted in more random patterns for some devices) was different for each individual device.

The experiments described in this appendix are still in a very preliminary stage and will be part of future work in the ongoing studies on this topic. What they do show however, is that the supply voltage curve on an SRAM can have a big impact on the behaviour of its start-up pattern. Therefore, it is be very important when implementing

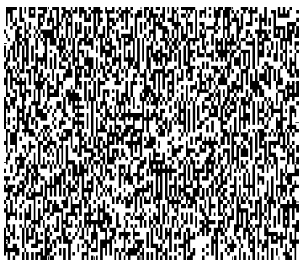

**Fig. 7.** Example start-up pattern of PIC16F1825 (1 KiB SRAM) with altered supply voltage power-up curve at +25 °C. White represents a bit with value 0, black a bit with value 1.

a secure PRNG seed generator as described in this paper to make sure that the supply voltage curve to the COTS microcontroller allows for good random noise behaviour in the SRAM start-up pattern.

Furthermore, based on these experiments we can think of at least one additional attack scenario: a hardware system that supplies a modified power-up curve to the microcontroller. It is possible on certain devices, with a specific power-up curve, to reduce the entropy contents of the SRAM to a minimum. Such an attack could be imagined on a cellphone, in which end-users can easily replace the battery, thus we call this the "evil battery" attack. Possible countermeasures are the use of power management and power monitoring circuits.

## B On the Selection of a Hash Function

In this section, we discuss the selection of a hash function implementation for secure seed generation. We make a suggestion for what is, in our opinion, the best choice, based on memory and code space efficiencies.

Any examples we give, we only give for the STM32F100R8. Due to the extremely low entropy in its SRAM start-up values (0.1% at −30 °C), the PIC16F1825 is not usable for secure PRNG generation using the suggested approach. Due to entropy being lost to temporary storage of the hash function, any conceivable hash function implementation will reduced the available entropy in the PIC16F1825 to 0.

Since it would fall outside the scope of this work, we did not implement any hash functions. Instead, we base our recommendations on Balasch et al. [2], in which a wide selection of 25 different hash functions, written in hand-optimized assembly, are presented. The implementations in [2] are for an 8-bit Atmel AVR chip, and thus not directly transferable to the chips that we have investigated. However, the required code sizes can be used as a relative size indicator. More important than code size though is the required amount of SRAM for temporary storage, which luckily is independent of the hardware architecture.

The required amount of SRAM for the hash function influences the total remaining entropy in the SRAM start-up pattern. No assumptions are made about the distribution of entropy within the SRAM, and therefore one has to assume a worst case scenario in which every byte of storage used by the hash function removes a full 8 bits of entropy from the total entropy available in the SRAM start-up pattern.

Assuming a required hash digest size of 256 bit, as required for FIPS 140-3 [16] compliance, the hash function, of those presented in [2], that requires the least storage is S-Quark, with 69 bytes. An alternative is PHOTON-256/32/32, which requires 82 bytes. If one prefers to use the newest SHA3 algorithm, then Keccak with parameters $r = 144, c = 256$ is the choice that requires the least amount of storage (114 bytes). Instead of using a dedicated hash function, it is also possible to use a block cipher-based hash construction. In that case, a Hirose/AES-256 construction is ideal, since it requires only 104 bytes of storage. These requirements, together with the remaining min-entropy in the STM32F100R8, are listed in Table 3.

**Table 3.** Hash function SRAM requirements and influence on SRAM entropy in STM32F100R8 (8 KiB SRAM). SRAM consumption data from [2]. Remaining min-entropy based on a pessimistic min-entropy estimate of 3% (see Section 3.3).

| **Hash** | **SRAM [byte]** | **Remaining min-entropy [bit]** |
|---|---:|---:|
| S-Quark | 69 | 1 414 |
| PHOTON-256/32/32 | 82 | 1 310 |
| Keccak[$r = 144, c = 256$] | 114 | 1 054 |
| Hirose/AES-256 | 104 | 1 134 |

It is obvious from Table 3 that the STM32F100R8 microcontroller has plenty entropy left over in its SRAM start-up pattern for any of the chosen hash function implementations. The formula used to calculate the remaining amount of entropy in SRAM for a given amount of memory consumption can be inverted to calculate the maximum allowed amount of SRAM consumption when requiring a minimum remaining entropy of 256 bit:

$$\begin{aligned} & 256 \leq 8 \cdot (8192 \cdot 0.03 - x) \\ \Leftrightarrow\ & x \leq 8192 \cdot 0.03 - \frac{256}{8} \\ \Leftrightarrow\ & x \leq 213.76, \end{aligned} \tag{1}$$

i.e. a hash function implementation on the STM32F100R8 can use a maximum of 213 bytes of SRAM when it is required that at least 256 bits of entropy remain under worst case conditions[7]. Thus, for the STM32F100R8, the choice of hash function will probably not need to depend on the used amount of SRAM, since enough entropy is likely available. For this microcontroller, the algorithm characteristics to look at then would be either required code side or execution time, depending on the application.

Note that, as mentioned in Section 4, block ciphers in CTR or OFB mode can be used as PRNGs. Thus, considering the benefits of code size reduction and a smaller

[7] Worst case conditions assume that every byte of SRAM used by the hash function reduces the total available entropy by 8 bits and that there is only 3% entropy in the SRAM start-up pattern of the STM32F100R8.

codebase to debug, we recommend the use of a block cipher both for hashing and as an PRNG. An obvious choice for a block cipher to use is AES, due to the facts that it is an internationally accepted standard, has been thoroughly studied and not been found vulnerable to attacks, and many optimized implementations exists for a wide range of platforms.

# Appendix B

# Helper Data Scheme Improvements

## B.1 Paper: “The Spammed Code Offset Method”

**Authors:** Boris Skoric, and Niels de Vreede
**Venue:** International Workshop on Trustworthy Embedded Devices (TrustED) 2013
**Date:** November 4th - 8th, 2013

**Status:** Pre-print of the paper available a the IACR ePrint archive (`http://eprint.iacr.org/2013/527`)

# The Spammed Code Offset Method

Boris Škorić and Niels de Vreede

## Abstract

*Helper data schemes are a security primitive used for privacy-preserving biometric databases and Physical Unclonable Functions. One of the oldest known helper data schemes is the Code Offset Method (COM). We propose an extension of the COM: the helper data is accompanied by many instances of fake helper data that is drawn from the same distribution as the real one. While the adversary has no way to distinguish between them, the legitimate party has more information and* can *see the difference. We use an LDPC code in order to improve the efficiency of the legitimate party's selection procedure.*
*Our construction provides a new kind of trade-off: more effective use of the source entropy, at the price of increased helper data storage. We give a security analysis in terms of Shannon entropy and order-2 Rényi entropy.*

## 1 Introduction

### 1.1 Helper Data Systems

The past decade has seen a lot of interest in a field that can be characterized as 'security with noisy data'. In several security applications it is necessary to *reproducibly* extract secret data from *noisy* measurements on a physical system. One such application is the privacy-preserving storage of biometric data. Analogously to password hashing, one can store biometric data in hashed form in order to prevent inside attackers from learning what the enrolled biometric features look like. Another application is read-proof storage of cryptographic keys using Physical Unclonable Functions (PUFs) [16, 17, 14]. Many types of digital memory can be considered insecure because of the large inbuilt redundancies needed to ensure reliable readout. PUFs provide an alternative way to store keys, namely in analog form, which allows the designer to exploit the inscrutability of analog physical behavior. Keys stored in this way are sometimes referred to as Physically Obfuscated Keys (POKs) [8].
In both the biometrics and the PUF/POK application, one faces the problem that some form of error correction has to be done, but under the constraint that the redundancy data, which is considered to be visible to attackers, does not reveal too much information about the secret extracted from the physical measurement. The problem is solved by a special security primitive, the *Helper Data System* (HDS).
A HDS in its most general form is shown in Fig. 1. The `Enroll` procedure takes as arguments a measurement $X$ and (optionally) a random value $R$. It outputs a secret $S$ and Helper Data $W$. The helper data is stored. In the reconstruction phase, a fresh measurement $X'$ is obtained. Typically $X'$ is a noisy version of $X$, i.e. close to $X$ but not necessarily identical. The `Rec` (reconstruction) procedure takes $X'$ and $W$ as input. It outputs $\hat{S}$, an estimate of $S$. If $X'$ is not too noisy then $\hat{S} = S$.

Two special cases of the general HDS are the Secure Sketch (SS) and the Fuzzy Extractor (FE) [6].

- The Secure Sketch has $S = X$ (and $\hat{S} = \hat{X}$, an estimator for $X$). If $X$ is not uniformly distributed, then $S$ is not uniform. The SS is suitable for privacy-preserving biometrics, where high entropy of $S$ (given $W$) is required, but not uniformness.

- The Fuzzy Extractor has a (nearly) uniform $S$ given $W$. The FE is typically used for extracting keys from PUFs and POKs.

There exists a generic construction to create a FE out of a SS: hashing the output of the SS using a Universal Hash Function (UHF) [3, 15, 11].

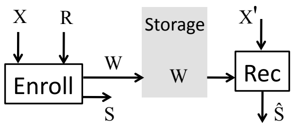

Figure 1: *Data flow in a generic Helper Data System.*

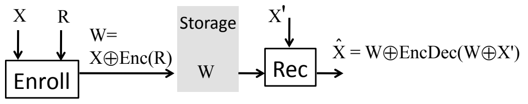

Figure 2: *The Code Offset Method employed as a Secure Sketch.*

### 1.2 The Code Offset Method

One of the oldest known SS constructions is the Code Offset Method (COM) [10, 6]. Here $X$ is a binary string, say of length $n$, with probability distribution $\rho$. The construction uses a linear error-correcting code that encodes $k$-bit messages as $n$-bit codewords. The encoding and decoding operations are denoted as `Enc` and `Dec` respectively. In Fig. 1 we take $R$ uniformly drawn from $\{0,1\}^k$ and

$$W = X \oplus \texttt{Enc}(R) \quad ; \quad \hat{X} = W \oplus \texttt{Enc}(\texttt{Dec}(W \oplus X')). \tag{1}$$

This is depicted in Fig. 2.
If $X$ is uniformly distributed on $\{0,1\}^n$ then the scheme is not only a SS but in fact also a FE; this holds because a uniform $X$ gives rise to helper data $W$ that leaks nothing about $R$. The formulas for the FE are: $S = R$ and $W = X \oplus \texttt{Enc}(R)$; $\hat{R} = \texttt{Dec}(X' \oplus W)$. Note that for uniform $X$ the $W$ reveals the syndrome of $X$, but nothing about $R$. Hence $R$ can then be used as a cryptographic key.
In this paper we study the case where $X$ is *not* uniformly distributed. A non-uniform $X$ appears naturally, e.g. in the

case of Coating PUFs [16], where Gray-coded capacitance measurements are concatenated to form $X$. Typically not all the Gray code words are represented, which leads to non-uniformity.
Similarly, a biometric feature vector is often split up into near-inependent components which each yield a small number of non-uniform bits.

## 1.3 Zero Leakage

For some sources $X$ it is possible to define helper data that reveals *nothing* about $S$. This is sometimes called Zero Secrecy Leakage (ZSL) helper data. The information contained in $X$ is split into two *independent* parts, one of which serves for error correction, and one for making the secret $S$. As we saw above, the COM with uniform $X$ has the ZSL property. Another example is the quantile partitioning scheme [18] of Verbitskiy et al. for *continuous* $X$, and its generalization to non-uniform $S$ [5].

## 1.4 Contributions and outline

In this paper we propose a simple modification of the Code Offset Method SS. The basic idea is to add a number (say $m-1$) of dummy helper data instances to the publicly stored enrollment data, and to randomly permute the list. All the instances are a priori indistinguishable from the point of view of the adversary, but the legitimate party possesses $X'$, which allows for efficient selection of the correct helper data instance. This workload asymmetry improves the security. For small $m$, the attacker may simply try out all possibilities, which leads to an average attack effort of $(m+1)/2$ times the original one. For very large $m$ this brute force attack is no longer feasible, and the attacker is forced to ignore the public data; in this way a new kind of 'zero leakage' is achieved, distinct from the ZSL of Section 1.3, namely public data that reveals practically nothing about $X$ (as opposed to $S$).
The concept of 'spamming' the attacker in this way is very general and is applicable whenever there exists an efficient way of recognizing the correct $W$ using $X'$. In this paper we show how the 'spamming' concept can be applied to the Code Offset Method. Our scheme requires the use of a linear error-correcting code with low-density parity check matrix (LDPC) in order to keep the legitimate party's workload low.
In Section 2 we introduce the notation and assumptions that we work with. In Section 3 we analyze the leakage of the ordinary Code Offset Method and briefly review the Leftover Hash Lemma. In Section 4 we present our new scheme, which we call the Spammed Code Offset Method (SCOM). Section 5 contains a security analysis of our scheme and a brief discussion of memory requirements. A discussion and conclusions are given in Section 6.

## 2 Notation and attacker model

Random variables are written with capitals, and their realizations in lower case. Vectors are in boldface; sets in calligraphic font. Concatenation is denoted as $||$. The notation $d_{\rm Hamm}(x,y)$ stands for the Hamming distance between $x$ and $y$. The logarithm 'log' is defined in base 2. The natural logarithm is ln.
The Code Offset Method works with a *linear* code $\mathcal{C}$ that has $n$-bit code words and $k$-bit messages. The encoding and decoding algorithms associated with this code are denoted as `Enc` and `Dec` respectively. The algorithm for computing the syndrome is denoted as `Syn`.
We consider a POK whose output at enrollment is a bit string $X \in \{0,1\}^n$. The probability distribution of $X$ is called $\rho$, and $\rho$ is not necessarily uniform. The string $R$ in Fig. 2 has length $k$. The helper data is called $W$. In the FE setting, the cryptographic key that is ultimately derived from the POK is denoted as $K \in \{0,1\}^\ell$.
We will use shorthand notation $p_{xw} = \Pr[X=x, W=w]$, $p_w = \Pr[W=w]$ and $p_{x|w} = \Pr[X=x|W=w]$, when it does not cause ambiguity. We define $q_z = \Pr[\texttt{Syn}(X)=z]$.
The public data stored in nonvolatile memory is $P$.
The outcome of the POK measurement in the reconstruction phase is denoted as $X' \in \{0,1\}^n$. The $X'$ is a noisy version of $X$, and in general does not have the same probability distribution as $X$. The estimator for $K$, derived from $X'$ and the public data, is denoted as $\hat{K}$.
We will rely on a cryptographic hash function $f$. Furthermore we will use a Universal Hash Function $g(x,a)$, where the second argument is public auxiliary randomness.
The attacker model is summarized as follows. We distinguish between two scenarios:

1. Biometric database for authentication.
   The adversary can read but not manipulate the public data $P$. His aim is to learn as much about $X$ as he can.[1]

2. Secure key storage with a POK.
   The adversary has access to the device which contains the POK. He cannot re-activate the device's enrollment mode of operation. The opacity of the POK, and the embedding of the POK in the device, prevent the adversary from reading out $K$ from the POK. Furthermore, physical tampering with the POK is unerringly detected by the device at the reconstruction phase. The public data $P$ is stored on the device in insecure nonvolatile memory. The adversary is able to read and to manipulate $P$. There is no Public Key Infrastructure that would allow the device to verify the authenticity of the public data. The adversary's main aim is to learn the POK key $K$. A secondary goal is to cause the device to accept a key other than $K$ as the correct key.

In both scenarios the adversary is able to discern whether reconstruction is successful. No other side channels exist.

## 3 Analysis of the Code Offset Method

We consider the general case of a non-uniform distribution $\rho$, and review what is known about the leakage of the COM. We briefly discuss the required amount of compression in case one wants to build a FE based on the COM.

[1] He may exploit this knowledge in various ways: (i) Some part of $X$ may reveal information about medical conditions. This is a privacy risk. (ii) Construct a fake biometric in order to pass authentication. This is a security risk. (iii) Cross-linking of people across different databases. This is a privacy risk.

**Lemma 1** *The Code Offset Method has the following probabilities,*

$$p_{rw} = 2^{-k}\rho(w \oplus \mathtt{Enc}(r)) \quad (2)$$

$$p_w = \frac{1}{2^k}\sum_{r\in\{0,1\}^k}\rho(w\oplus\mathtt{Enc}(r)) = \frac{1}{2^k}\Pr[\mathtt{Syn}X = \mathtt{Syn}\,w] \quad (3)$$

$$p_{r|w} = \frac{\rho(w \oplus \mathtt{Enc}(r))}{\sum_{t\in\{0,1\}^k}\rho(w \oplus \mathtt{Enc}(t))} \quad (4)$$

$$p_{xw} = 2^{-k}\rho(x)\delta_{\mathtt{Syn}(w),\mathtt{Syn}(x)} \quad (5)$$

$$p_{x|w} = \frac{\rho(x)\delta_{\mathtt{Syn}(w),\mathtt{Syn}(x)}}{\sum_{t\in\{0,1\}^k}\rho(w \oplus \mathtt{Enc}(t))}. \quad (6)$$

Proof: The $R$ is drawn uniformly and thus each $r$ has probability $2^{-k}$ of occurring. The probability $\Pr[W = w|R = r]$ is given by $\rho(w \oplus \mathtt{Enc}(r))$. Multiplication of these two gives (2). Equation (3) follows by computing $p_w$ as the marginal of $p_{rw}$ by summing over $r$. Eq. (4) follows from $p_{r|w} = p_{rw}/p_w$. Finally, (5) and (6) follow from (2) and (4) by setting $x = w \oplus \mathtt{Enc}(r)$, which is only possible if $x$ and $w$ have the same syndrome. □

Eq. (4) shows that in general $p_{r|w} \neq p_r$. Thus, $W$ leaks information not only about $\mathtt{Syn}(X)$, but also about $R$.

**Lemma 2** *In the Code Offset Method it holds that*

$$\mathsf{H}(W) = k + \mathsf{H}(\mathtt{Syn}\,X) \quad (7)$$

$$\mathsf{H}(X,W) = \mathsf{H}(X) + k \quad (8)$$

$$I(X;W) = \mathsf{H}(\mathtt{Syn}\,X). \quad (9)$$

Proof: $\mathsf{H}(W)$ follows directly from (3), and $\mathsf{H}(X,W)$ from (5). The $I(X;W)$ is computed as $\mathsf{H}(X)+\mathsf{H}(W)-\mathsf{H}(X,W)$. □

In order to obtain a nearly uniform key $K$ from $X$ (or, equivalently, from $R$), one has to hash down to a smaller size (say $\ell$): $K = g(X,A) \in \{0,1\}^\ell$. Here $A$ is *public* auxiliary randomness that serves as a 'catalyst' for the UHF $g$.

Let $U$ be a uniform variable on $\{0,1\}^\ell$. The relation between $\ell$ and the uniformity of $K$ is given by the Leftover Hash Lemma (LHL) [9] and can be formulated as

$$\ell \leq L_\varepsilon(X,W) \implies \mathbb{E}_w[\Delta(U;K|W=w)] \leq \varepsilon \quad (10)$$

$$\text{with } L_\varepsilon(X,W) = \mathsf{H}_2(X|W) + 1 - 2\log\frac{1}{\varepsilon}. \quad (11)$$

Eq. (10) states that the non-uniformity of $K$ given $W$ does not exceed $\varepsilon$ as long as $X$ has been sufficiently hashed down. The $\ell$ must not exceed the '$\varepsilon$-extractable randomness' $L_\varepsilon$. The notation $\mathsf{H}_2$ in (11) stands for the conditional Rényi entropy of order two and is defined as [7]

$$\mathsf{H}_2(X|W) = -2\log Q_2(X|W)$$

$$Q_2(X|W) = \mathbb{E}_w\sqrt{\textstyle\sum_x p_{x|w}^2} = \sum_w\sqrt{\textstyle\sum_x p_{xw}^2}, \quad (12)$$

where $\mathbb{E}_w$ stands for the expectation value over $W$.

Note that the 'penalty' term $2\log\frac{1}{\varepsilon}$ in (11) depends only on $\varepsilon$, i.e. it depends not on the *improvement* of the uniformity but on the *final* uniformity. Because of this fact, the approach using UHFs can be quite wasteful.

Remark: Under some conditions [1] the factor 2 in the penalty term can be replaced by 1. Furthermore, the LHL can be sharpened a bit by considering *smooth* Rényi entropy [12, 13, 19]. Such details are beyond the scope of the current paper.

## 4 Our construction: the Spammed Code Offset Method

We first show a naive spamming approach, without efficient de-spamming at the reconstruction phase. Then we propose an efficient scheme, in two variants: one in the privacy-preserving biometrics context, the other in the secure key storage context.

The efficient scheme requires a linear block code with a *low-density* parity check matrix. The security and the storage requirements are analyzed in Section 5.

### 4.1 Naive approach

**Algorithm E0: Enrollment, the naive way**

```
1. Measure $X \in \{0,1\}^n$.
2. Draw $R \in \{0,1\}^k$ uniformly at random.
3. Compute helper data $W = X \oplus \mathtt{Enc}(R)$.
4. For $j \in \{1,\dots,m-1\}$ do:
    (a) Uniformly draw $\Sigma_j \in \{0,1\}^k$.
    (b) Draw $D_j \in \{0,1\}^n$ from the distribution $\rho$.
    (c) Compute $\Omega_j = D_j \oplus \mathtt{Enc}(\Sigma_j)$.
5. Draw a random permutation $\pi$.
6. Construct a vector $\boldsymbol{\Omega} = \pi(\Omega_1,\cdots,\Omega_{m-1},W)$.
7. Compute $G = f(\boldsymbol{\Omega}||X)$.
8. Store public data $P = (\boldsymbol{\Omega}, G)$.
```

**Algorithm R0: Reconstruction, the naive way**

```
1. Read $P' = (\boldsymbol{\Omega}', G')$.
2. Measure $X'$.
3. Set $\mathcal{L}_1 = \emptyset$. For $j \in \{1,\dots,m\}$ do:
    (a) Try to compute $R_j = \mathtt{Dec}(X' \oplus \Omega'_j)$.
    (b) If the decoding succeeds then add $j$ to $\mathcal{L}_1$.
4. If $\mathcal{L}_1 = \emptyset$ then abort.
5. Set $\mathcal{L}_2 = \emptyset$. For $i \in \mathcal{L}_1$ do:
    (a) $\hat{X}_i = \Omega'_i \oplus \mathtt{Enc}(R_i)$
    (b) Compute $G_i = f(\boldsymbol{\Omega}'||\hat{X}_i)$.
    (c) If $G_i = G'$ then add $i$ to the list $\mathcal{L}_2$.
6. If $|\mathcal{L}_2| \neq 1$ then abort; else $\hat{X} = X_{\mathcal{L}_2}$.
```

Enrollment steps 2 and 3 represent the construction of the ordinary COM helper data $W$. The $D_j$ in step 4b are decoy measurements. They follow the same distribution as $X$ and are therefore statistically indistinguishable from a real POK measurement. The $\Omega_j$ is the COM helper data associated with the decoy $D_j$. The purpose of the random permutation in step 6 is to hide from the adversary which entry of $\boldsymbol{\Omega}$ is the actual helper data. Here it is crucial that the adversary cannot 'see inside' the hash $G$. Since the entries of $\mathbf{\Omega}$ are distributed exactly in the way real helper data should be, his knowledge about the statistics of $W$ does not help him to decide which entry is the real one. This intuitive statement is made more precise in Section 5.

Note that in step 7 the hash is computed over the *whole* vector $\mathbf{\Omega}$. This ensures that any manipulation of the public data will be detected, be it in the hash, in $W$ or in the decoys. This way of protecting the helper data against manipulation was introduced by [2]. Alternatively, one may use a Message Authentication Code with Key Manipulation Security [4].

At reconstruction the public data may have been altered, which is why we use the notation $P'$, $\mathbf{\Omega}'$, $G'$ in step 1 of algorithm R0. A list $\mathcal{L}_1$ is made of $\mathbf{\Omega}'$ entries that lead to successful decoding. The whole set $\mathcal{L}_1$ has to taken into account, since some of the decoys may by chance decode, and the order of the entries is random. The list of candidates is further narrowed down to a list $\mathcal{L}_2$ of entries whose $X_j$ generates the correct hash. If $P' = P$ and $X' \approx X$ then typically there is only one candidate left in $\mathcal{L}_2$. If $P' \neq P$ or $X'$ is too far away from $X$ to be error-corrected, then typically $\mathcal{L}_2 = \emptyset$. (Algorithm R0 continues the search after having found its first match. Alternatively, we could stop.)

The main idea behind the scheme is that the adversary cannot distinguish between the true helper data and the decoys, while the device's knowledge of $X'$ helps it to see the difference.

It may happen that the choice of system parameters is such that R0 has a long running time. For instance, the processing time in step 3 is linear in $m$, where $m$ may be a large number. Furthermore, the choice of $n$, $k$, and $m$ may give rise to a long list $\mathcal{L}_1$, and the number of hashes that has to be computed in step 5 is linear in $|\mathcal{L}_1|$. In the schemes below we aim to reduce the running time of the reconstruction algorithm.

## 4.2 Scheme #1: Secure Sketch for biometrics database

Below we show a more efficient pair of algorithms, in the biometrics scenario. The main difference with the naive approach is the use of the syndromes $\mathbf{\Phi}$. Note that $\texttt{Syn}(X) = \texttt{Syn}(W)$. Hence revealing $F$ conveys no extra information to the adversary; he could already compute $F$ from $W$ unaided.

The idea behind scheme #1 is that comparing $\texttt{Syn}(X')$ to $\texttt{Syn}(X)$ and the other syndromes allows the device to heuristically re-order $\mathbf{\Omega}'$ in such a way that the most likely candidates are tried out first. Here it is crucial that the parity check matrix of the code has low density: then a small Hamming distance between $X$ and $X'$ leads to a small Hamming distance between $\texttt{Syn}(X)$ and $\texttt{Syn}(X')$.

The running time of the reconstruction algorithm R1 is practically independent of the number of dummies, except for steps 4–6 which are lightweight. Most importantly, due to the sorting R1 does not have to compute many decodings and hashes.

**Algorithm E1:**
**enrollment for biometrics database**

```
1. Measure the biometric $X \in \{0,1\}^n$.
2. Draw $R \in \{0,1\}^k$ uniformly at random.
3. Compute the syndrome $F = \texttt{Syn}(X)$.
   Compute helper data $W = X \oplus \texttt{Enc}(R)$.
4. For $j \in \{1, \ldots, m-1\}$ do:
   (a) Uniformly draw $\Sigma_j \in \{0,1\}^k$.
   (b) Draw $D_j \in \{0,1\}^n$ from the distribution $\rho$.
   (c) Compute $\Phi_j = \texttt{Syn}(D_j)$.
       Compute $\Omega_j = D_j \oplus \texttt{Enc}(\Sigma_j)$.
5. Choose a random permutation $\pi$.
6. Construct a vector $\mathbf{\Phi} = \pi(\Phi_1, \cdots, \Phi_{m-1}, F)$.
   Construct a vector $\mathbf{\Omega} = \pi(\Omega_1, \cdots, \Omega_{m-1}, W)$.
7. Compute $G = f(\mathbf{\Phi}||\mathbf{\Omega}||X)$.
8. Store public data $P = (\mathbf{\Phi}, \mathbf{\Omega}, G)$.
```

**Algorithm R1: efficient biometric verification**

```
1. Read $P' = (\mathbf{\Phi}', \mathbf{\Omega}', G')$.
2. Measure the fresh biometric $X'$.
3. Compute $F' = \texttt{Syn}(X')$.
4. For $j \in \{1, \ldots, m\}$ do: $d_j = d_{\text{Hamm}}(F', \Phi'_j)$.
5. Make a permutation $\lambda$ that sorts $(d_j)_{j=1}^m$ in ascending order.
6. Let $\tilde{\mathbf{\Omega}} = \lambda(\mathbf{\Omega}')$.
7. Let $j = 0$.
8. Increase $j$. If $j = m+1$ then abort.
9. Try to compute $R_j = \texttt{Dec}(X' \oplus \tilde{\Omega}_j)$.
   If the decoding fails then goto 8.
10. $\hat{X}_j = \tilde{\Omega}_j \oplus \texttt{Enc}(R_j)$.
11. If $G' \neq f(\mathbf{\Phi}'||\mathbf{\Omega}'||\hat{X}_j)$ then goto 8.
12. Accept.
```

*Remark 1*: In step 7 of E1, the $\mathbf{\Phi}$ and $\mathbf{\Omega}$ also serve as salt for the hashing of $X$.

*Remark 2*: There are many alternative ways to organize the steps in (R1,E1). For instance, in step 6 of R1 the vector $\mathbf{\Omega}'$ does not have to be physically permuted; permutation of the indices $\{1, \cdots, m\}$ is more efficient.

## 4.3 Scheme #2: Fuzzy Extractor

Below we present the Fuzzy Extractor version of algorithms (E1, R1), in the POK scenario.

**Algorithm E2: enrollment for POK**

```
1. Measure the POK output $X \in \{0,1\}^n$.
2. Generate random $A$. Compute $K = g(X, A)$.
3. Draw $S \in \{0,1\}^k$ uniformly at random.
4. Compute the syndrome $F = \texttt{Syn}(X)$.
   Compute helper data $W = X \oplus \texttt{Enc}(S)$.
5. For $j \in \{1, \dots, m-1\}$ do:
   (a) Uniformly draw $\Sigma_j \in \{0,1\}^k$.
   (b) Draw $D_j \in \{0,1\}^n$ from the distribution $\rho$.
   (c) Compute $\Phi_j = \texttt{Syn}(D_j)$.
       Compute $\Omega_j = D_j \oplus \texttt{Enc}(\Sigma_j)$.
6. Choose a random permutation $\pi$.
7. Construct a vector $\mathbf{\Phi} = \pi(\Phi_1, \cdots, \Phi_{m-1}, F)$.
   Construct a vector $\mathbf{\Omega} = \pi(\Omega_1, \cdots, \Omega_{m-1}, W)$.
8. Compute $G = f(\mathbf{\Phi}||\mathbf{\Omega}||A||X)$.
9. Store public data $P = (\mathbf{\Phi}, \mathbf{\Omega}, A, G)$.
```

**Algorithm R2: efficient reconstruction of POK**

```
1. Read $P' = (\mathbf{\Phi}', \mathbf{\Omega}', A', G')$.
2. Measure the POK output $X'$.
3. Compute $F' = \texttt{Syn}(X')$.
4. For $j \in \{1, \dots, m\}$ do: $d_j = d_{\text{Hamm}}(F', \Phi'_j)$.
5. Make a permutation $\lambda$ that sorts $(d_j)_{j=1}^m$ in ascending order.
6. Let $\tilde{\mathbf{\Omega}} = \lambda(\mathbf{\Omega}')$.
7. Let $j = 0$.
8. Increase $j$. If $j = m+1$ then abort.
9. Try to compute $R_j = \texttt{Dec}(X' \oplus \tilde{\Omega}_j)$.
   If the decoding fails then goto 8.
10. $\hat{X}_j = \tilde{\Omega}_j \oplus \texttt{Enc}(R_j)$.
11. If $G' \neq f(\mathbf{\Phi}'||\mathbf{\Omega}'||A'||\hat{X}_j)$ then goto 8.
12. $\hat{K} = g(\hat{X}_j, A')$.
```

The only difference with the biometrics scenario (E1,R1) is the use of the auxiliary randomness $A$ and the computation of $K$ and $\hat{K}$.

## 5 Analysis of the SCOM

We investigate how much information about $X$ is revealed to the adversary by showing him $\mathbf{\Omega}$. In principle we should be looking at the leakage from the whole public data $P$, but there one hits a snag: information-theoretically there is no such a thing as a one-way function. The hash $G$ hides its input *in practice*, but information theory gives $I(X;P) = I(X;W)$. The leakage from $\mathbf{\Omega}$ is a better way to represent the adversary's actual workload.
In the biometrics scenario, the relevant quantity to look at is Shannon entropy. (One might argue that min-entropy is more important, but since we do not have the stringent requirements that cryptographic keys have to satisfy[2] , we will stick to Shannon entropy.) The relevant quantity in the POK scenario is the Rényi entropy $\mathsf{H}_2$, which features in the $\varepsilon$-extractable randomness (11). We show results for both scenarios.
In Section 5.3 we also briefly look at memory requirements.

### 5.1 Leakage in terms of Shannon entropy

We first present two lemmas that allow us to relate the leakage $I(X;\mathbf{\Omega})$ to the adversary's ignorance about the permutation $\Pi$. Then we present a result for small $m$ and for large $m$.

**Lemma 3** *The adversary's ignorance about $X$ given $\mathbf{\Omega}$ can be written as*

$$\mathsf{H}(X|\mathbf{\Omega}) = \mathsf{H}(X|W) + I(\Pi; X\mathbf{\Omega}). \tag{13}$$

<u>Proof:</u> We write $\mathsf{H}(X|\mathbf{\Omega}\Pi)$ in two ways: as $\mathsf{H}(X|W)$ and as $\mathsf{H}(X\Pi|\mathbf{\Omega}) - \mathsf{H}(\Pi|\mathbf{\Omega}) = \mathsf{H}(X\Pi|\mathbf{\Omega}) - \mathsf{H}(\Pi) = \mathsf{H}(X|\mathbf{\Omega}) + \mathsf{H}(\Pi|X\mathbf{\Omega}) - \mathsf{H}(\Pi)$. Equating the two different expressions yields (13). □

**Lemma 4** *Let $t(x, \boldsymbol{\omega})$ denote the number of entries in $\boldsymbol{\omega}$ that are consistent with $x$, i.e. $t(x, \boldsymbol{\omega}) = |\{j : \texttt{Syn}(\omega_j) = \texttt{Syn}(x)\}|$. Then*

$$\mathsf{H}(X|\mathbf{\Omega}) = \mathsf{H}(X|W) + \log m - \mathbb{E}_{x\boldsymbol{\omega}} \log t(x, \boldsymbol{\omega}). \tag{14}$$

<u>Proof:</u> We write $I(\Pi; X\mathbf{\Omega}) = \mathsf{H}(\Pi) - \mathsf{H}(\Pi|X\mathbf{\Omega}) = \log m! - \mathsf{H}(\Pi|X\mathbf{\Omega})$. For a given $t(x, \boldsymbol{\omega})$ there are $t$ possible ways to place $w$ in $\boldsymbol{\omega}$; furthermore there are $m-1$ further entries to be permuted, which can be done in $(m-1)!$ ways. The total number of permutations $\pi$ consistent with $x$ and $\boldsymbol{\omega}$ is $t \cdot (m-1)!$. They are all equiprobable from the point of view of the adversary. Hence $\mathsf{H}(\Pi|X = x, \mathbf{\Omega} = \boldsymbol{\omega}) = \log[t \cdot (m-1)!]$. It follows that $\mathsf{H}(\Pi|X\mathbf{\Omega}) = \mathbb{E}_{x\boldsymbol{\omega}} \log[t \cdot (m-1)!] = \log(m-1)! + \mathbb{E}_{x\boldsymbol{\omega}} \log t$ and $I(\Pi; X\mathbf{\Omega}) = \log \frac{m!}{(m-1)!} - \mathbb{E}_{x\boldsymbol{\omega}} \log t$. Finally we substitute this expression for $I(\Pi; X\mathbf{\Omega})$ into Lemma 3. □

**Theorem 1** *The conditional entropy $\mathsf{H}(X|\mathbf{\Omega})$ can be bounded from below as*

$$\mathsf{H}(X|\mathbf{\Omega}) \geq \mathsf{H}(X|W) + \log m - \frac{m-1}{\ln 2} \mathbb{E}_x q_{\texttt{Syn}(x)}. \tag{15}$$

<u>Proof:</u> We write $t(x, \boldsymbol{\omega}) = 1 + u(x, \boldsymbol{\omega})$ and use $\ln(1+u) \leq u$. This gives $\mathbb{E}_{x\boldsymbol{\omega}} \log t(x, \boldsymbol{\omega}) \leq \frac{1}{\ln 2} \mathbb{E}_{x\boldsymbol{\omega}} u(x, \boldsymbol{\omega})$. For given $x$, the $u$ is binomial-distributed with parameters $m-1$ and $q_{\texttt{Syn}(x)}$. (See Section 2 for the notation $q$.) Thus we have $\mathbb{E}_{x\boldsymbol{\omega}} u(x, \boldsymbol{\omega}) = \mathbb{E}_x (m-1) q_{\texttt{Syn}(x)}$. □
At first sight (15) might seem to contradict the well known principle 'conditioning reduces Shannon entropy'. However, it should be borne in mind that $\mathbf{\Omega}$ is not just $W$ plus decoys; $\mathbf{\Omega}$ results from a *function* $\Pi$ applied to $W$ and the decoys. The function $\Pi$ reduces the leaking effect of $W$.

[2]Remember that most biometrics cannot be kept secret, since it is possible to measure them surreptitiously.

The probability $q_{\texttt{Syn}(x)}$ is typically of the order $1/2^{n-k}$ if $X$ is not too strangely distributed. Hence the last term in (15) is a small correction term if $m < 2^{n-k}$. Eq. (15) confirms the intuitive idea that the attacker's effort increases by a factor $\approx m/2$. Note that the bound in Theorem 1 is far from tight when $m$ is large. For large $m$ we have the following result.

**Theorem 2** *The conditional entropy $\mathsf{H}(X|\boldsymbol{\Omega})$ can be bounded from below as*

$$\mathsf{H}(X|\boldsymbol{\Omega}) \geq \mathsf{H}(X) - \frac{1}{m}\cdot\frac{2^{n-k}-1}{\ln 2}. \qquad (16)$$

Proof: As in the proof of Theorem 1, we write $t = 1 + u$. Furthermore we split $u$ into its expectation value (at fixed $x$) and a deviation: $u = (m-1)q_{\texttt{Syn}(x)} + \delta$, where $\mathbb{E}_\delta \delta = 0$. This gives

$$\begin{aligned}\mathbb{E}_{x\boldsymbol{\omega}} \log t &= \mathbb{E}_x \log[m q_{\texttt{Syn}(x)}] + \mathbb{E}_x \log[1 + \frac{1-q_{\texttt{Syn}(x)}}{m q_{\texttt{Syn}(x)}}] \\ &\quad + \mathbb{E}_{x\delta} \log(1 + \frac{\delta}{1+[m-1]q_{\texttt{Syn}(x)}}). \qquad (17)\end{aligned}$$

Next we use $\ln(1+z) \leq z$ twice, and $\mathbb{E}_\delta\delta = 0$, to get

$$\begin{aligned}\mathbb{E}_{x\boldsymbol{\omega}} \log t &\leq \log m + \mathbb{E}_x \log q_{\texttt{Syn}(x)} + \frac{1}{\ln 2}\mathbb{E}_x \frac{1-q_{\texttt{Syn}(x)}}{m q_{\texttt{Syn}(x)}} \\ &= \log m + \sum_{z\in\{0,1\}^{n-k}} q_z \log q_z + \frac{1}{m\ln 2}(-1 + \sum_z \frac{q_z}{q_z}) \\ &= \log m - \mathsf{H}(\texttt{Syn}X) + \frac{2^{n-k}-1}{m\ln 2}. \qquad (18)\end{aligned}$$

Substitution of (18) into Lemma 4 finishes the proof. $\Box$

If $m$ is of order $2^{n-k}$ or larger, then the $\frac{1}{m}$ term in Theorem 2 is a small correction term; we see that $\boldsymbol{\Omega}$ then hardly leaks anything about $X$, as we expected intuitively.

## 5.2 Leakage in terms of Rényi entropy

We present a bound on $\mathsf{H}_2(X|\boldsymbol{\Omega})$ that is useful for large $m$. We observe that for the adversary each entry in $\boldsymbol{\omega}$ is equally likely to be the correct one. Thus, his knowledge about $x$ can be parametrized as a probability distribution that is conditioned on each of the entries $\omega_j$ with equal probability $1/m$. This gives

$$p_{x|\boldsymbol{\omega}} = \frac{1}{m}\sum_{j=1}^{m} p_{x|\omega_j}. \qquad (19)$$

Based on (19) we obtain the following result.

**Theorem 3** *The conditional Rényi entropy $\mathsf{H}_2(X|\boldsymbol{\Omega})$ can be bounded from below as*

$$\mathsf{H}_2(X|\boldsymbol{\Omega}) \geq \mathsf{H}_2(X) - \frac{1}{m\ln 2}\left[\frac{\mathbb{E}_w \sum_x p^2_{x|w}}{\sum_x p^2_x} - 1\right]. \qquad (20)$$

Proof:

$$\begin{aligned}\mathsf{H}_2(X|\boldsymbol{\Omega}) &= -2\log \mathbb{E}_{\boldsymbol{\omega}} \sqrt{\textstyle\sum_x p^2_{x|\boldsymbol{\omega}}} \qquad (21)\\ &\geq -2\log\sqrt{\mathbb{E}_{\boldsymbol{\omega}} \textstyle\sum_x p^2_{x|\boldsymbol{\omega}}} \qquad (22)\\ &= -\log \mathbb{E}_{\boldsymbol{\omega}} \sum_x \frac{1}{m^2}\sum_{a,b=1}^{\ell} p_{x|\omega_a} p_{x|\omega_b} \qquad (23)\\ &= -\log \frac{1}{m^2}\sum_x \left[\sum_a \mathbb{E}_{\boldsymbol{\omega}} p^2_{x|\omega_a} + \sum_{a,b:a\neq b} \mathbb{E}_{\boldsymbol{\omega}} p_{x|\omega_a}p_{x|\omega_b}\right] \qquad (24)\\ &= -\log \sum_x \left[\frac{1}{m}\mathbb{E}_w p^2_{x|w} + [1-\frac{1}{m}]p^2_x\right] \qquad (25)\\ &= -\log\left[(\sum_x p^2_x)(1 + \frac{\mathbb{E}_w \sum_x p^2_{x|w} - \sum_x p^2_x}{m\sum_x p^2_x})\right] \qquad (26)\\ &= \mathsf{H}_2(X) - \log(1 + \frac{\mathbb{E}_w \sum_x p^2_{x|w} - \sum_x p^2_x}{m\sum_x p^2_x}). \qquad (27)\end{aligned}$$

In (22) we used Jensen's inequality. In (23) we substituted (19). Finally (20) is obtained from (27) by using $\log(1+z) = \ln(1+z)/\ln 2 \leq z/\ln 2$. $\Box$

Remark: If $X$ is not too far from uniform, then the $\frac{1}{m}$-term in Theorem 3 is of order $2^{n-k}/m$, i.e. the same order of magnitude as the $\frac{1}{m}$-term in Theorem 2.

When spammed helper data $\boldsymbol{\Omega}$ is used instead of $W$, the entropy $\mathsf{H}_2(X|W)$ in the extractable randomness formula (11) can be replaced by the (much) larger number $\mathsf{H}_2(X|\boldsymbol{\Omega})$.

## 5.3 Storage requirements

In the biometrics scenario there is a large amount of storage space per enrolled person, since the public data $P$ is usually stored in a dedicated database. Blowing up the database by a factor $m$ could be feasible. Furthermore, the original $W$ is a very small thing to start with.

In the POK scenario, however, the public data is usually stored on the device that contains the POK. This device has to be cheap; hence nonvolatile memory may become an issue.

Below we tabulate some numerical estimates. The $k = 128$ case corresponds to a POK with a 128-bit key. The $k = 64$ case represents the biometrics scenario. (We could even have chosen $k$ a little smaller.) The 'err' is the number of errors that the LDPC code can correct. Under 'Mem' we list the space required to store $\boldsymbol{\Omega}$ and $\boldsymbol{\Phi}$, namely $m\cdot(2n-k)$ bits. The values of $n$ are approximate and are meant only to give orders of magnitude. For $m$ we list two values: $m = 2^{(n-k)/2}$, which reduces the leakage from $W$ by approximately half the bits, and $m = 2^{n-k}$ which almost completely cancels the leakage.

| | $k = 64$ | | | $k = 128$ | | |
|---|---|---|---|---|---|---|
| err | $n$ | $\log m$ | Mem | $n$ | $\log m$ | Mem |
| 1 | 72 | 4 | 0.2KB | 138 | 5 | 0.6KB |
| | | 8 | 2.5KB | | 10 | 19KB |
| 2 | 78 | 7 | 1.4KB | 146 | 9 | 10KB |
| | | 14 | 0.2MB | | 18 | 5.1MB |
| 3 | 85 | 10.5 | 19KB | 154 | 13 | 0.2MB |
| | | 21 | 27MB | | 26 | 1.4GB |

**Table 1:** *Memory required to store $\boldsymbol{\Omega}$ and $\boldsymbol{\Phi}$, listed as a function of $k$, the number of errors to be corrected and* $\log m$.

## 6 Discussion

We have proposed the Spammed Code Offset Method, in which the adversary gets spammed with bogus helper data. For small spam factor $m$, the security is increased by roughly $\log m$ bits. For large $m$, of the order $2^{n-k}$ or larger, the leakage $I(X;W)$ is practically eliminated. These statements are quantified in Theorems 1, 2 and 3. While the workload of the adversary is increased, the workload of the legitimate party (counting only calls to `Dec` and the hash function $f$) stays almost constant as a function of $m$. This is achieved by using Hamming distance in syndrome space as a fast candidate selection criterion, where the use of an LDPC code makes sure that a small distance between $X$ and $X'$ translates to a small distance in syndrome space.

In the POK scenario it depends on various system parameters whether it makes sense to use the SCOM. If the available nonvolatile memory in the device is limited and there is ample entropy in $X$, then the ordinary COM suffices. Table 1 illustrates that for a 128-bit key and $\log m$ comparable to $n-k$, the size of the public data rapidly becomes infeasibly large as the number of errors increases. However, it should be borne in mind that a even a small $m$ already yields a (modest) security improvement. The SCOM provides a new kind of trade-off: a more effective use of source entropy is achieved at the price of digital memory usage.

In the biometrics case it is especially important to eliminate the leakage $I(X;W)$, since the entropy of $X$ is usually rather low and has to be maximally exploited. Fortunately it is easier to meet the memory requirements in this scenario.

As future work we will do experiments with various LDPC codes in order to optimize the search in algorithms R1 and R2, and in order to get more precise numbers in Table 1. Another interesting issue to look at is the cross-linkability between biometric templates in different databases. When lots of decoy entries are added to the templates, it becomes much harder for an adversary to decide if templates in different databases belong to the same person, since the decoys are likely to cause false matches.

**Acknowledgements**
We thank Ruud Pellikaan, Ludo Tolhuizen, Joep de Groot and Jean-Paul Linnartz for useful discussions.

## References

[1] B. Barak, Y. Dodis, H. Krawczyk, O. Pereira, K. Pietrzak, F.-X. Standaert, and Y. Yu. Leftover Hash Lemma, revisited. In *CRYPTO*, volume 6841 of *LNCS*, pages 1–20. Springer, 2011.

[2] X. Boyen, Y. Dodis, J. Katz, R. Ostrovsky, and A. Smith. Secure remote authentication using biometric data. In *Eurocrypt 2005*, volume 3494 of *LNCS*, pages 147–163. Springer-Verlag, 2005.

[3] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[4] R. Cramer, Y. Dodis, S. Fehr, C. Padró, and D. Wichs. Detection of algebraic manipulation with applications to robust secret sharing and fuzzy extractors. In *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 471–488, 2008.

[5] J.A. de Groot, B. Škorić, N. de Vreede, and J.-P. Linnartz. Information leakage of continuous-source Zero Secrecy Leakage Helper Data Schemes. `http://eprint.iacr.org/2012/566`, 2012.

[6] Y. Dodis, M. Reyzin, and A. Smith. Fuzzy Extractors: How to generate strong keys from biometrics and other noisy data. In *Eurocrypt 2004*, volume 3027 of *LNCS*, pages 523–540. Springer-Verlag, 2004.

[7] S. Fehr and S. Berens. The conditional Rényi entropy, 2013.

[8] B. Gassend. Physical Random Functions. Master's thesis, Massachusetts Institute of Technology, 2003.

[9] J. Håstad, R. Impagliazzo, L.A. Levin, and M. Luby. Construction of pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[10] A. Juels and M. Wattenberg. A fuzzy commitment scheme. In *ACM Conference on Computer and Communications Security (CCS) 1999*, pages 28–36, 1999.

[11] J.-P. Kaps, K. Yüksel, and B. Sunar. Energy scalable universal hashing. *IEEE Trans. Computers*, 54(12):1484–1495, 2005.

[12] R. Renner and S. Wolf. Smooth Rényi entropy and applications. In *IEEE International Symposium on Information Theory (ISIT) 2004*, page 233, 2004.

[13] R. Renner and S. Wolf. Simple and tight bounds for information reconciliation and privacy amplification. In *Asiacrypt 2005*, volume 3788 of *LNCS*, pages 199–216. Springer-Verlag, 2005.

[14] A.-R. Sadeghi and D. Naccache, editors. *Towards hardware-intrinsic security*. Springer, 2010.

[15] D.R. Stinson. Universal hashing and authentication codes. *Designs, Codes, and Cryptography*, 4:369–380, 1994.

[16] P. Tuyls, G.-J. Schrijen, B. Škorić, J. van Geloven, R. Verhaegh, and R. Wolters. Read-proof hardware from protective coatings. In *Cryptographic Hardware and Embedded Systems (CHES) 2006*, volume 4249 of *LNCS*, pages 369–383. Springer-Verlag, 2006.

[17] P. Tuyls, B. Škorić, and T. Kevenaar. *Security with Noisy Data: Private Biometrics, Secure Key Storage and Anti-Counterfeiting*. Springer, London, 2007.

[18] E.A. Verbitskiy, P. Tuyls, C. Obi, B. Schoenmakers, and B. Škorić. Key extraction from general nondiscrete signals. *IEEE Transactions on Information Forensics and Security*, 5(2):269–279, 2010.

[19] B. Škorić, C. Obi, E. Verbitskiy, and B. Schoenmakers. Sharp lower bounds on the extractable randomness from non-uniform sources. *Information and Computation*, 209:1184–1196, 2011.