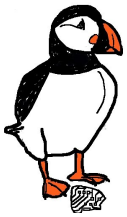

PUFFIN

Physically unclonable functions found in standard PC components

Project number: 284833
FP7-ICT-2011-C

D1.2

Scientific contribution of WP1, part 2 Exploration

Due date of deliverable: 31. January 2015
Actual submission date: 3. March 2015

WP contributing to the deliverable: WP1

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 WP1, part 2
Exploration

Pol van Aubel (external: Radboud Universiteit Nijmegen)
Daniel J. Bernstein (TUE)
Anthony van Herrewege (KUL)
Tanja Lange (TUE)
Ruben Niederhagen (TUE)
André Schaller (TUD)
Christian Schlehuber (TUD)
Peter Simons (IID)

3. March 2015
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 summarizes the scientific contribution of Work Package 1 (WP1) during the second phase (months 19–36) of the PUFFIN project, and presents an overall perspective on the accomplishments of WP1.

Keywords: WP1, exploration, uninitialized SRAM, cleared SRAM, tough PUFs, microcontrollers, smartphones, GPUs, CPUs, testbeds, reusability

Contents

List of Figures

Chapter 1

Introduction

WP1, the Exploration work package, searches for new ways to physically identify PCs and other commodity hardware. It focuses on standard PCs, handheld devices, and embedded systems as they actually exist today and in the foreseeable future. The goal is not to modify components to make them easy to identify; the goal is to find identifiers that are already intrinsic in existing mass-market hardware. The most productive explorations have been of uninitialized SRAM, discussed below in more detail.

WP2 is responsible for comprehensive data analysis. However, WP1 is responsible for making a preliminary assessment of the quality of data obtained from hardware, so that WP2 can focus on the most interesting data.

Early in the PUFFIN project, upon request from WP3, WP1 broadened its explorations to search for randomness, and for nondeterministic behavior in general, even if the source of randomness does not seem suitable for use as an identifier. For the case of SRAM this means that even small amounts of uninitialized SRAM are potentially useful, even if those amounts are clearly not enough to support robust identification.

WP1 has successfully read out uninitialized memory from a surprisingly wide range of processors, ranging from tiny embedded processors up to graphics cards costing 500 EUR. This document describes WP1's exploration of these processors, and in particular the challenges that WP1 faced in accessing the memory on these processors.

1.1 Exploration vs. utilization

A common theme in WP1's SRAM investigations is that, after finding a potentially usable bank of SRAM, WP1 builds tools to copy the SRAM to a general-purpose laptop CPU for further analysis.

These short-term tools should not be confused with the more complex long-term tools produced by WP3. In some cases the short-term tools have been a starting point for building the long-term tools, but the primary purpose of the short-term tools is for WP1 to collect data, while the primary purpose of the long-term tools is to provide security for various applications. In many cases the long-term tools, when installed, *prevent* exactly the types of copying performed by the short-term tools.

1.2 First phase vs. second phase

The work in PUFFIN was formally divided into two phases: phase 1 (months 1 through 18) and phase 2 (months 19 through 36). "D1.1: Scientific contribution of WP1, part 1" (23 pages) was a preliminary report on WP1's work in the first phase.

The main function of D1.2, this document, is to report WP1's work in the second phase. To keep this document self-contained, and to avoid the need for readers to consult two overlapping documents, WP1 has merged the contents of D1.1 into this document, producing a unified description of WP1's work throughout the PUFFIN project. The advances from the first phase to the second phase are summarized as part of Section 1.3.

1.3 Measurements vs. tools

Most of the *measurements* and *successes* from WP1 are naturally presented in the context of analyses and applications, and are correspondingly integrated into reports from WP2 and WP3. For example, Chapter 2 of D2.2 lists more than ten types of devices analyzed by WP2, and of course the measurements of those devices were produced by WP1. In particular, most of the new second-phase analyses reported there began with second-phase work in WP1: two new microcontrollers (TI Stellaris LM4F120H5QR and Atmel ATmega1280); one new smartphone-type CPU (TI Sitara AM3358, the same ARM Cortex-A8 microarchitecture used in the iPad 1 and iPhone 4 in 2010 and subsequently reused in lower-cost devices); many more GTX 295 multiprocesors; temperature variations for three platforms; etc. Similarly, the successful applications reported in D3.2 rely on PUFs found by WP1.

This document instead focuses on the *tools* and *techniques* developed in WP1. The externally visible results of WP1 are the discoveries of many new PUFs (and new entropy sources), but the internal results of WP1 are advances in the discovery *process*, particularly the process of locating uninitialized SRAM in a wide range of platforms:

- WP1 included small embedded devices (see Chapter 2) as additional targets. Microcontroller CPUs are the simplest widespread processors and have the fewest obstacles to accessing uninitialized SRAM, making them a good **testbed** for tool development. These CPUs are also worthy of study in their own right: embedded devices are increasingly important as security platforms (consider, e.g., wirelessly reconfigurable pacemakers), and the same CPUs often reappear as components of larger devices. This work started during the first phase; it was not included in the original project plan.

- WP1 systematically tackled the challenges posed by more complex platforms. Sometimes SRAM is initialized by hardware, making it useless for PUFFIN; but sometimes SRAM is a "tough PUF", uninitialized by hardware but initialized by another *software* component such as a device driver, the operating-system kernel, the bootloader, or even the BIOS. Distinguishing these two situations requires **taking control** of each platform at the deepest possible level, extracting SRAM data before the data can be ruined by other software. This line of exploration for GPUs (see Chapter 4) was the original proof of concept for PUFFIN. The first phase of the project included analogous exploration of smartphones (see Chapter 3) and deeper exploration of GPUs. The second phase of the project pushed those platforms further and extended this exploration to laptop/desktop CPUs (see the paper in Appendix A).

- WP1 built frameworks to ensure that the WP1 techniques are, and will be, **reused** for as many devices as possible. Increased automation of the measurement process allows systematic measurement of many devices: for example, WP1 measured 510 "streaming multiprocessors" on 17 separate graphics chips (collecting 19 measurements from each multiprocessor, with 16360 bytes in each measurement). Crowdsourcing (not written up yet; work in progress) allows the Internet community to contribute measurements of many more devices. Documentation and open-source software allow the same tools to be adapted to other platforms, both today and in the future. A small part of WP1 activity in the first phase of the PUFFIN project, and a much larger part in the second phase, was aimed at reusability beyond the project.

Chapter 2

Microcontroller SRAM

In this chapter, we first describe the methods used to extract SRAM start-up data from four different types of microcontrollers: Microchip PIC16F1825, STMicro STM32F100R8, Texas Instruments MSP430F5308, and Atmel ATMega328p. Afterwards, we give details about the SRAM extraction specific to each type of microcontroller.

2.1 Firmware design

Our general strategy for copying uninitialized SRAM data out of a microcontroller is as follows. The microcontroller is 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 enters an idle loop. Care is taken in this firmware to avoid overwriting any of the SRAM storage. This is easy to achieve on microcontrollers that have several general-purpose registers to store variables, such as a pointer to the current SRAM byte. However, some microcontrollers, such as the Microchip PIC16 family, have only a single general-purpose register. To avoid overwriting SRAM on these microcontrollers, we store some variables in unused configuration registers.

2.2 Hardware setup

We obtained initial measurements of SRAM power-up patterns as follows. We manually connected the power lines and serial port of the target microcontroller to an external serial TTL-to-USB converter, and connected the converter to a self-powered USB hub. After taking an SRAM measurement, we switched off power to the microcontroller (i.e., left the power floating) for at least 10 seconds. The goal of this discharging period is to ensure that the microcontroller has discharged completely and that the SRAM will contain fresh data on the next power-up. However, for some devices, this discharging period is insufficient: it is important to connect the power supply lines to ground in order to completely discharge any remaining energy within the microcontroller.

In order to extract start-up patterns more reliably and efficiently, we created a custom measurement board meeting the following requirements:

1. Allow connection of many microcontrollers at once.
2. Be extensible with regard to the number of attached microcontrollers.

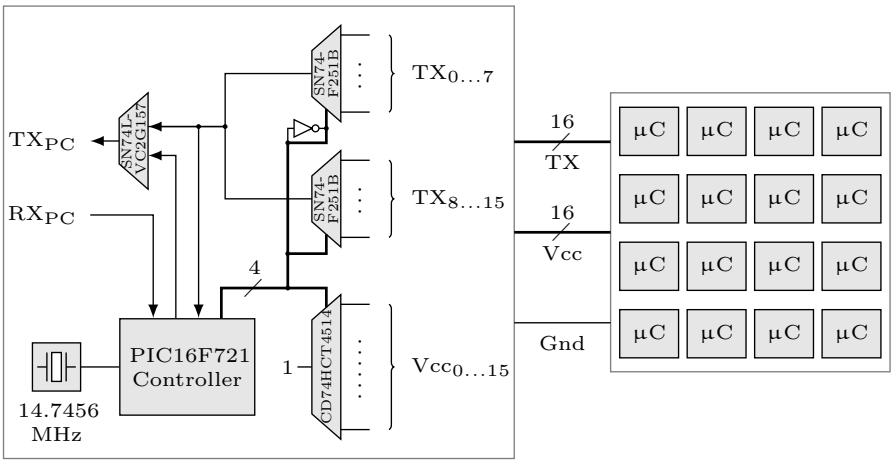

Figure 2.2.1: High-level schematic of the measurement controller board (*left*) with a board of microcontrollers to be measured attached (*right*).

3. Support remote setup.
4. Make automated, unsupervised measurements possible.
5. Support any realistic baud rate.
6. Support an 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. We used an oscilloscope to verify requirement 7 for our controller board; note that this is important in order to generate realistic start-up patterns. Finally, the demultiplexer on our controller board connects non-active power lines to ground, meeting requirement 8; note that this is important in order to erase the state of the SRAM completely on power-down. A simplified schematic of our design is shown in Fig. 2.2.1.

Central to the board is a PIC16F721 microcontroller which drives a 4-to-16 demultiplexer as well as two 8-to-1 multiplexers. Due to the low current requirements of the devices being measured, the outputs of the demultiplexer can be used as power supply lines. Each of the multiplexer inputs is connected to the serial transmissions port of one of the attached microcontrollers. Furthermore, a 2-to-1 multiplexer is included to allow the controller to switch serial output between either its own serial port or that of the currently powered microcontroller. Since there are some unused pins left on the PIC16F721 this design could

be extended with the help of some logic gates to allow the connection of up to at least 1024 microcontrollers.

As noted earlier, the devices should be completely discharged internally in order to get fresh SRAM power-up values. Otherwise, remnants of previously stored values might linger in memory. Thus, simply disconnecting a microcontroller from its power supply is not sufficient to ensure valid measurements during the next power-up cycle. This makes the selection of the demultiplexer crucial to being able to take valid measurements quickly. We therefore choose to use a CD74HCT4514 demultiplexer, because it connects non-selected outputs to ground, thereby discharging the attached microcontroller.

Keeping in mind future extensibility, it is preferable for the multiplexers to have a tri-state output. This allows wiring together the output of multiple multiplexers, of which only one has its output enabled. Unfortunately, 16-to-1 three-state multiplexers are not produced any more and thus we have chosen to use two 8-to-1 three-state multiplexers, more specifically the SN74F251.

The serial interface speed of the PIC16F721 controller should match that of the devices being measured. Therefore, even though the PIC16F721 has an internal 16 MHz oscillator, it is clocked by an external UART clock (i.e., 14.7456 MHz). This allows the baud rate of the PIC16F721 controller to be adapted on-the-fly to the baud rate of the microcontrollers being measured.

In order to create a flexible measurement platform which can handle any number of microcontrollers with SRAM of any size, the serial output of the current microcontroller being measured is fed into the PIC16F721 measurement controller. After the power supply for a microcontroller has been enabled and that microcontroller has been given some time to power up, the controller starts checking the serial output. If the output remains idle for too long, then either the SRAM measurement is finished or no microcontroller is available at the currently selected position, and the controller advances to the next microcontroller. This system allows for fast, repeated, unattended measurements in which measurement times are automatically adapted to allow full SRAM extraction to take place without requiring any configuration changes to the controller.

For each family of microchips to be measured, a custom PCB was designed containing just the microcontrollers and a minimum of external components (e.g., LEDs to allow debugging feedback and decoupling capacitors), thereby eliminating the change of external components interfering with the microcontroller start-up sequence.

A photograph of the measurement board attached to a prototype PCB for STM32F100R8 readout is shown in Figure 2.2.2.

In order to easily program the surface-mounted microcontrollers we built a device that we call a “programming pen”. This pen is attached to the microcontroller programmer and connects to the target IC using six pogo pins. On the target PCB, a small footprint of 6 vias is required to mate with these pogo pins. Additionally, we added an USB interface that can be used to command a PC to program the microcontroller at the press of a button embedded into the programming pen. An photograph of this device can be seen in Fig. 2.2.3.

2.3 Details on specific microcontrollers

In this section we will outline the details specific to each microcontroller family which had to be taken into account to be able to extract the complete SRAM power-up data.

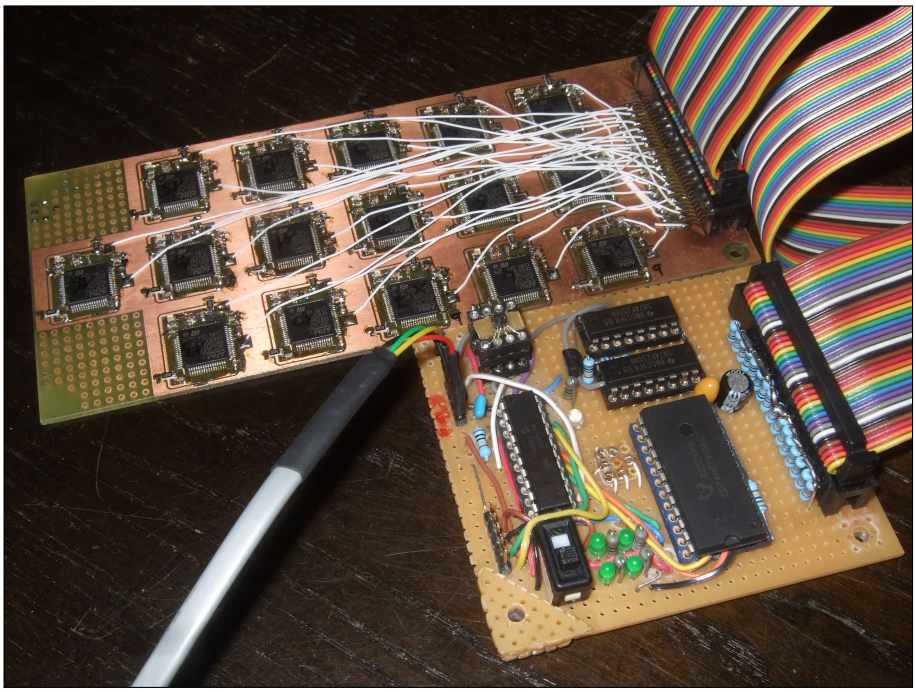

Figure 2.2.2: Measurement controller (*bottom*) connected to prototype STM32F100R8 board (*top*).

2.3.1 Microchip PIC16F1825

The Microchip PIC16 family is peculiar in that it has only a single working register. Futhermore, due to its 8-bit architecture, it requires banking in order to address the full address space. The last 16 elements of the general purpose SRAM are mapped back to bank 0. Due to this banking, the complete general purpose SRAM, which is what we want to extract, has non-sequential addresses. Furtunately, newer PIC16 architectures, such as the PIC16F1825 which we use, have a seperate linear mapping for these SRAM sections. This linear mapping excludes the shared 16 elements, so those have to be handled separately. In our extraction firmware, we first extract the shared 16 elements, and then loop over the linear mapped SRAM.

WP2 reported surprisingly low entropy in our first measurements from the PIC16F1825 chips, so we tried multiple variations of voltage curves on startup. For one such voltage curve, shown in Fig. 2.3.1 on the right, the SRAM contents turned out to have slightly higher entropy. Unfortunately, Microchip does not wish to provide any details on the internal silicon layout of their chips, so it is quite difficult to figure out what causes these effects.

We also noted that all Microchip PIC16F devices we tested kept their SRAM values for over 10 minutes when their power supply line was left floating. This observation was previously reported in dedicated SRAM devices [**?**], but never observed in COTS microcontrollers. Our custom measurement board eliminated this issue, ensuring proper discharge and a fresh SRAM power-up state.

Figure 2.2.3: Programming pen used to program microcontrollers through a set of pogo pins.

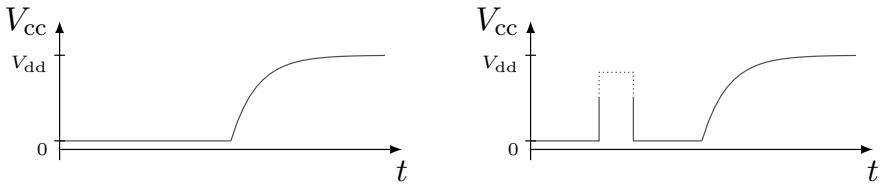

Figure 2.3.1: Powerup voltage supply variation tests on PIC16F1825. Original supply voltage curve (*left*, normal for SRAMs) and altered curve (*right*).

2.3.2 STMicro STM32F100R8

The STM32F100R8 has a 32-bit ARM Cortex-M3 architecture. We extracted SRAM by simply looping over and reading out a linear address range.

2.3.3 Texas Instruments MSP430F5308

The MSP430F5308 has a 16-bit architecture. No banking is required; we extracted SRAM by looping over and reading out a linear address range, as on the STM32F100R8.

2.3.4 Atmel ATMega328p

The ATMega328p is used on the very popular Arduino development boards. It has an 8-bit architecture. As with the two previous chips, we extracted SRAM by looping over a linear address range.

Chapter 3

Smartphone SRAM

Smartphones are much more complex devices than the microcontrollers considered in Chapter 2. WP1 decided to start its smartphone explorations with a reasonably well documented development board, the PandaBoard (ES). This board contains the same TI OMAP4460 system-on-chip used in many smartphones, and multimedia capabilities similar to modern smartphones; it was designed by TI and is sold to the general public with support from a TI subsidy.

The PandaBoard has two ARM Cortex-A9 cores; two Cortex-M3 cores for signal processing; and 2 gigabytes of DDR memory. It also contains the following on-chip memory (OCM) instances:

- 4096 bytes of "Save-and-Restore ROM", presumably not useful.
- 8192 bytes of "Save-and-Restore RAM".
- 57344 bytes of "L3 OCM RAM".

WP1 explored possible accesses to the different memory instances. Analysis showed uninitialized SRAM in the L3 OCM RAM. However, further analysis also showed that the 57344 bytes are not completely usable for fingerprint extraction since a fraction of the memory region is pre-initialized, presumably by the board's ROM code. Reading the first 13312 bytes of the L3 OCM RAM at an early stage of the boot process produced a unique SRAM start-up pattern. To achieve this, WP1 modified the bootloader (u-boot). The modification of the bootloader consisted of finding the appropriate code position for adding the read-out code such that no previous code interacted with the target memory region and thus overwrote the initial SRAM values. The added code loops a pointer through the memory region, displaying each byte on the bootloader's console for retrieval by a controlling PC for further analysis.

Chapter 4

GPU SRAM

Many users of desktop computers, laptop computers, tablets, and smartphones spend the bulk of their processing power on computer graphics, notably as a major component of video games. The only way for a chip manufacturer to provide competitive performance for these applications is to devote large amounts of chip area to the operations used in these applications: for example, heavily vectorized low-precision floating-point multiplications.

Chip manufacturers have, however, been hesitant to include these features in mass-market general-purpose CPUs. Chip area is not free; devoting large amounts of chip area to video games means taking the same area away from features that are critical for many other important CPU applications.

These pressures created a market for add-on "graphics processing units" (GPUs). All users have CPUs; many users add GPUs to provide extra processing power for computer graphics; graphics applications are designed to offload appropriate computations from the CPU to the GPU. In recent years CPU designers have begun to offer integrated chips, with varying numbers of CPU cores and GPU cores on each chip, but the GPU cores are still designed as separate special-purpose cores devoted to graphics applications.

The PUFFIN team already identified GPUs in 2010 as a possible source of uninitialized SRAM visible directly to applications. There are several relevant differences between large CPUs and GPUs:

- Large CPUs evolved various reliability and security features to support multiuser operating systems, often handling critical and sensitive data. GPUs evolved as single-user special-purpose processors, and are generally perceived as handling nothing more than video-game data.

- In particular, large CPUs include "virtual memory" providing a separate address space for each application and "memory protection" separating multiple users of the same computer. Typical GPUs do not (yet) provide either of these features.

- Allowing programs to directly read SRAM after reset could compromise CPU memory protection, so it is unsurprising for a large CPU to disable access to SRAM after reset, taking this concern away from the OS. Typical GPUs have relatively large amounts of SRAM and have no comparable reason to clear the SRAM after reset.

- Large CPUs use SRAM primarily as a cache for DRAM. Typical GPUs expose SRAM and DRAM directly to the programmer, limiting the chip area required for cache logic.

These differences provide reasons to hope that uninitialized SRAM will be more easily visible on GPUs than it is on CPUs on the same computers.

On the other hand, GPU hardware is much more poorly documented than CPU hardware. The GPU SRAM is not directly accessible by the CPU through the PCI bus; the CPU programs the GPU to access data and copy it to the CPU. GPU programing normally uses semi-portable high-level interfaces such as OpenGL, CUDA, and OpenCL; the actual low-level hardware interface is hidden behind compilers and device drivers. NVIDIA's PTX "assembly language" is actually another semi-portable high-level language, hiding most of the hardware details.

The PUFFIN proposal reported that the PUFFIN team had successfully accessed the power-on state of 1.25% of the SRAM from two NVIDIA GPUs (60 "multiprocessors" in two physical chips), in total 30720 bytes from each GPU (1024 bytes from each multiprocessor). This work took advantage of a new assembly language developed by TUE for NVIDIA Tesla-architecture GPUs, providing much more control than NVIDIA's lowest-level programming language.

After the PUFFIN project started, WP1 successfully accessed a larger fraction of the SRAM from the GPUs, and then developed a new SRAM readout tool using NVIDIA's PTX "assembly language". PTX is really another semi-portable high-level language, hiding most of the hardware details, but provides just barely enough control to access specified locations in SRAM. The resulting main loop is very simple:

```
__global__ void doit(int *results,int words)
{
  for (int i = 0;i < words / THREADS;++i) {
    int pos = threadIdx.x + i * THREADS;
    int data;
    asm("ld.shared.s32 %0, [%1];" : "=r"(data) : "r"(pos << 2));
    results[blockIdx.x * words + pos] = data;
  }
}
```

The power-on SRAM contents appear to contain large amounts of random data. Powering off and on again produces a similar, but not identical, SRAM state. Overwriting the SRAM state and resetting the GPU again produces a similar state, as if the SRAM state had never been overwritten. A different GPU has a different power-on SRAM state. These observations were consistent with what one would expect from uninitialized SRAM.

These explorations encountered a new challenge when WP1 upgraded to the latest versions of the NVIDIA GPU drivers. These drivers appear to clear large amounts of GPU SRAM, presumably in an effort to reduce the amount of undocumented behavior exposed to GPU applications. However, the drivers do not clear SRAM bytes at positions 32 through 63 on each GPU core. Each of the 60 multiprocessors thus provided 32 bytes of easily accessible uninitialized SRAM data. WP1 measured this data across a series of power-off/pause/power-on/measure cycles, and forwarded the results to WP2.

During the second phase of the project, WP1 drastically improved its GPU SRAM extraction capability, successfully reading a considerable fraction of physical SRAM: 16360 bytes out of the 16384 bytes of "shared memory" in each multiprocessor (out of the 81920 bytes of total SRAM), i.e., 490800 bytes out of the 491520 bytes of shared memory in each GPU. WP1

improved the automation of its measurement system, obtained access to 17 “identical” multiprocessors, measured each of the multiprocessors across a series of 19 power-off/pause/power-on/measure cycles, and again forwarded the results to WP2.

Bibliography

[1] Sergei Skorobogatov. Low Temperature Data Remanence in Static RAM. Technical report, University of Cambridge, June 2002. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf.

Appendix A

Desktop/laptop CPU SRAM

A.1 Paper: "Investigating SRAM PUFs in AMD64 CPUs"

Authors: Pol van Aubel and Ruben Niederhagen
Venue: To be determined
Date: —
Status: Plan to extend paper to include results from more platforms.

Investigating SRAM PUFs in AMD64 CPUs

Pol van Aubel[1] and Ruben Niederhagen[2]

[1] Radboud University Nijmegen
Digital Security Group
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
radboud@polvanaubel.com
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
ruben@polycephaly.org

Abstract. Physically unclonable functions (PUFs) provide data that can be used for cryptographic purposes: on the one hand randomness for the initialization of random-number generators; on the other hand individual fingerprints for unique identification of specific hardware components. As of today, for off-the-shelf personal computers, randomness and individual fingerprints are provided only in form of additional or dedicated hardware. This paper investigates the presence of intrinsic PUFs in AMD64 CPUs of off-the-shelf personal computers. In particular we investigate registers as potential PUF sources in the operating-system kernel, the bootloader, and the system BIOS. We also consider the cache in the early boot stages. In all cases we encounter negative results, i.e., we are not able to derive any PUF, because the hardware appears to initialize all SRAM before it is made available for software access.

Keywords: Physically unclonable functions, SRAM PUFs, randomness, hardware identification.

1 Introduction

The most common CPU architectures for computing devices like notebooks, desktop computers, and servers are the x86 and AMD64 architectures. The AMD64 architecture, also known as x86-64 and x64, was introduced by AMD in 1999 as a backwards-compatible successor to the pervasive x86 architecture. For networked operation in the Internet, which is a hostile environment, these devices need a multitude of cryptographic primitives, e.g., cryptographic operations with secret keys, keyed hash functions, secure randomness, and, in some cases, remote attestation and identification capabilities. In this paper we focus on two seemingly conflicting aspects: The generation of *random bit strings*, which requires indeterministic behaviour, and the generation of *unique identifiers*, which requires deterministic behaviour.

Randomness is required for several purposes in cryptography. For example, random bit sequences are used to generate secret encryption keys and nonces in cryptographic protocols in order to make them impossible for an attacker to guess. Many cryptographic primitives assume the presence of a secure random source; however, CPUs are designed to be deterministic and sources of randomness are rare.

Unique identifiers can be used to deterministically derive an identity-based cryptographic key. This key can be used for authentication and data protection. For example, it is possible to use this key for hard disk encryption: The hard drive, i.e., the bootloader, operating system, and user data, are encrypted with this secret intrinsic key and can only be decrypted if the unique identifier is available. The identifier thus must be protected from unauthorized access.

Currently, these features are provided by accompanying the device with dedicated hardware: randomness is offered, e.g., by the `RDRAND` hardware random number generator; identification, e.g., by a Trusted Platform Module (TPM). However, these solutions can only be used if a dedicated

TPM is available in the device or if the CPU supports the `RDRAND` instruction which only recently was introduced with Intel's Ivy Bridge CPUs. Furthermore, they do not help in cases where the cryptographic key should be bound to the identity of the CPU itself.

However, for these cryptographic functionalities additional hardware is not necessarily required: randomness as well as identification can be derived from individual physical characteristics inherent to a silicon circuit by the use of physically unclonable functions (PUFs). PUFs can be derived from, e.g., ring oscillators [1], signal delay variations [2, 3], flip-flops [4], latches [5], and static random-access memory (SRAM) [6, 7]. While most of these require dedicated circuits, SRAM is available in most CPUs in form of registers and caches.

SRAM PUFs were initially identified in FPGAs. The PUF characteristics of SRAM are derived from the uninitialized state of SRAM immediately after power-up. When unpowered SRAM cells are powered up, they obtain a value of 0 with a certain probability P_0, or 1 with probability $P_1 = 1 - P_0$. The individual probabilities of each SRAM cell depend on minor manufacturing differences and are quite stable over time. Some of the cells have a probability close to 1 for either P_0 or P_1 and thus tend to give the same value at every power-up. Because of this stability, and because the pattern of this stability is different for every block of SRAM, they can be used for fingerprinting. Other cells have a probability close to 0.5 for both P_0 and P_1 and thus tend to give a different value at each power-up. Since their behavior is unstable, they are a good source for randomness.

Before the power-up state of SRAM can be used as PUF, an enrollment phase is required: the SRAM is powered up several times in order to measure which SRAM cells are suitable for randomness and which for fingerprinting. For the actual use of the SRAM PUF some postprocessing is performed, e.g., a feedback loop can be used in order to avoid bias in the generated random bit sequence and an error correction code in order to compensate for occasional bit errors in the fingerprint.

At TrustED 2013, researchers demonstrated in [8] that SRAM-based PUFs exist in various brands of popular microcontrollers, such as AVR and ARM, which are commonplace in mobile and embedded devices. More recently [9] used this to secure a mobile platform. Intrinsic SRAM PUFs have also been identified in GPUs [10].

Since SRAM is used in abundance in the caches and registers of AMD64 CPUs, they may carry intrinsic PUFs. In [11] the authors propose an instruction-set extension to utilize this SRAM to build a secure trusted computing environment within the CPU. However, research on existing PUFs in AMD64 CPUs appears non-existent. The obvious question is whether such PUF capabilities are currently also exhibited by (i.e., available and accessible in) x86 and AMD64 CPUs. The documentation of these processors contains a number of statements which suggest that — even though such SRAM PUFs may exist — they are impossible to access from software running on those CPUs (see Section 3).

This paper investigates whether it is indeed impossible to use registers and caches of AMD64 CPUs as PUFs. The result of our investigation is a negative one, in the sense that for the specific CPU we investigated fully (an AMD E350) we have to confirm that even at the earliest boot stages we cannot use registers or caches as PUFs.

To enable reproducibility of our results, and to allow other researchers to investigate other CPUs, we place all our modifications to the software described in this paper into the public domain. The source code and patches are available at `https://gist.github.com/anonymous/3de7705681689626d4f7`.

This paper is structured as follows: In the next section, we describe our experimental setup, i.e., the AMD64 processor architecture and our test mainboard, the ASRock E350M1. In Section 3 we describe how we investigate if registers can be accessed sufficiently early in the boot process in order to read their power-on state and use them as SRAM PUFs. In Section 4 we investigate the suitability of the processor cache as SRAM PUF during BIOS execution when the processor is in the *cache-as-RAM* mode. Finally, in Section 5 we discuss our results.

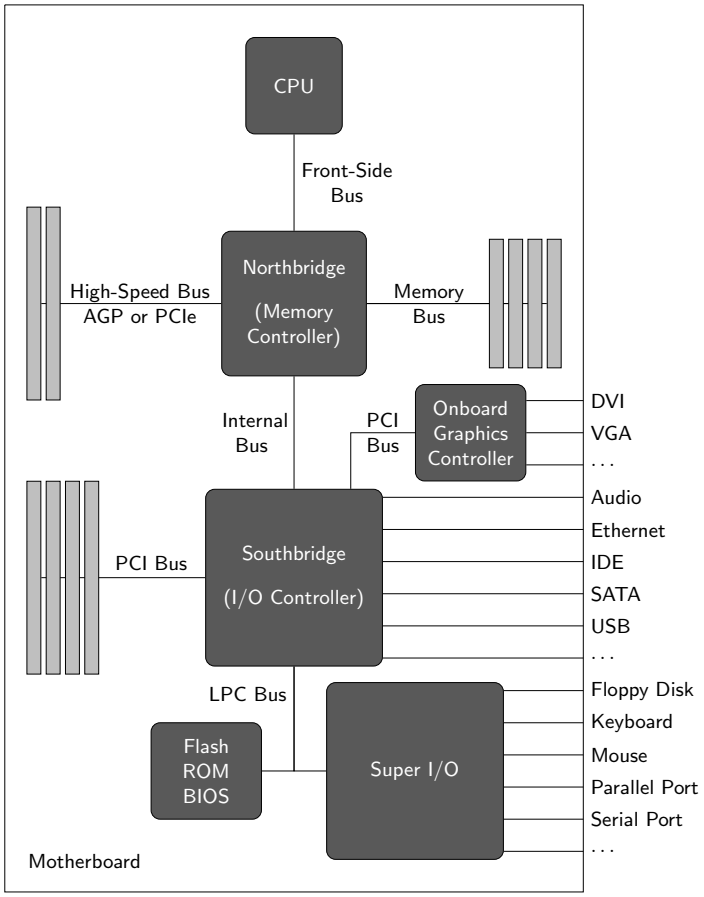

Fig. 1: Schematic of the AMD64 motherboard architecture.

2 Experimental setup

Our main experimental setup consisted of a single mainboard with an AMD64 CPU.

AMD64 architecture. Computers based on the AMD64 architecture have a long history, tracing back to the IBM PC. The most common setup, visualized in Figure 1, is based on a motherboard that has a socket for an AMD64 architecture CPU, a memory controller and slots for Random Access Memory, several communication buses such as PCI and PCI Express and associated slots for expansion cards, non-volatile memory for storing the system's boot firmware, and a "chipset" tying all these together. This chipset consists of a Northbridge, handling communication between the CPU and high-speed peripherals such as graphics hardware and main memory, and the Southbridge, handling everything else, with the Northbridge as an intermediary to the CPU.

Finally, there is the Super I/O chip. This chip condenses many I/O features which were traditionally handled by different circuits into one chip. This is the reason that the current iteration of AMD64 motherboards still supports many features found on boards from 20 years ago, such as serial port I/O, floppy-disk drives, and parallel ports, next to relatively new features such as Serial ATA and PCI Express. However, some of these features might not be exposed to the user: The Super I/O chip that is used to drive these subsystems often supports the entire range of "old" functionalities, but only those which the motherboard manufacturer deems worthwhile to offer are actually exposed through sockets on the board. The serial port, for example, is still exposed as a header on most boards, or at least as a solder-on option. Since these are relatively simple I/O devices, they are often the first to be initialized after system startup and can be used for output of, e.g., system diagnostics during the early boot stage before the graphics hardware has been initialized.

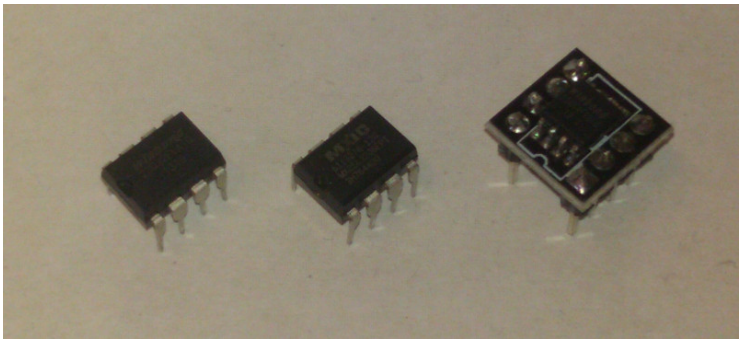

Fig. 2: Chips used on the E350M1 motherboard. Left: the original Winbond 25Q32FVAIQ. Center: The unsuitable replacement MX25L3206EPI. Right: The working replacement Winbond 25Q64FVSIG

In recent years, functions the Northbridge used to handle, such as memory control and graphics-hardware control, were integrated into the CPU. This was done to reduce overhead and speed limitations caused by having to go through an intermediary chip. Since this lifted most of the high-speed demands from the Northbridge, this development has caused manufacturers to integrate the few remaining functions of the Northbridge and the functions of the Southbridge into a single chip. The main principles of operation of the motherboard, however, remain the same.

Test mainboard. Our main test board is the E350M1, manufactured by ASRock. On it runs an AMD E-350 APU (Accelerated Processing Unit, a package embedding a CPU and graphics controller) which was first manufactured in 2011, with an AMD A50M chipset. It has an exposed serial port header and a socketed 4 MiB Winbond 25Q32FVAIQ NVRAM chip for the UEFI or BIOS firmware. The board has on-board flash capabilities for this chip. The form factor is mini-ITX. The E-350 APU itself has two processor cores, with 32 KiB level-1 data cache, 32 KiB level-1 instruction cache, and 512 KiB of level-2 cache per core.

As explained later in Section 3.4, the main reasons for picking this mainboard are that it supports a fairly recent AMD CPU, has a socketed NVRAM chip, and is supported by the open-source BIOS implementation coreboot [12].

The integration of graphics hardware, combined with the small form factor, make this a board suited for general-purpose home computing and multimedia computers.

We acquired two sets of replacement NVRAM chips. The first set consisted of five MXIC MX25L3206EPI. These chips closely match the original chip's specifications, yet are from a different manufacturer. They failed to boot the board with anything other than the original UEFI firmware. The second set consisted of two Winbond 25Q64FVSIG chips. These chips are almost identical to the original, with only two major differences: they have twice the storage size (8 MiB), and a different form factor (SOIC8 instead of DIP8). Therefore, they required an adapter circuit to fit the form factor. However, these chips served the purpose of booting the board with modified firmware. The three different types of chips can be seen in Figure 2. For flashing these chips under Linux, we used the open-source software flashrom.

For mass storage (bootloader and operating system) we used a simple USB stick. For I/O we used a normal setup of keyboard, mouse and screen, but also attached a serial socket to the serial port header, and used a serial-to-USB adapter to get serial output from BIOS and bootloader. The test setup can be seen in Figure 3.

Finally, power was supplied by a normal ATX power supply, and we powered, unpowered and reset the board by shorting the corresponding pins with a metal tab. Measurements were taken by manually powercycling the board and reading the measurement output from screen (kernel) or serial output (BIOS and bootloader).

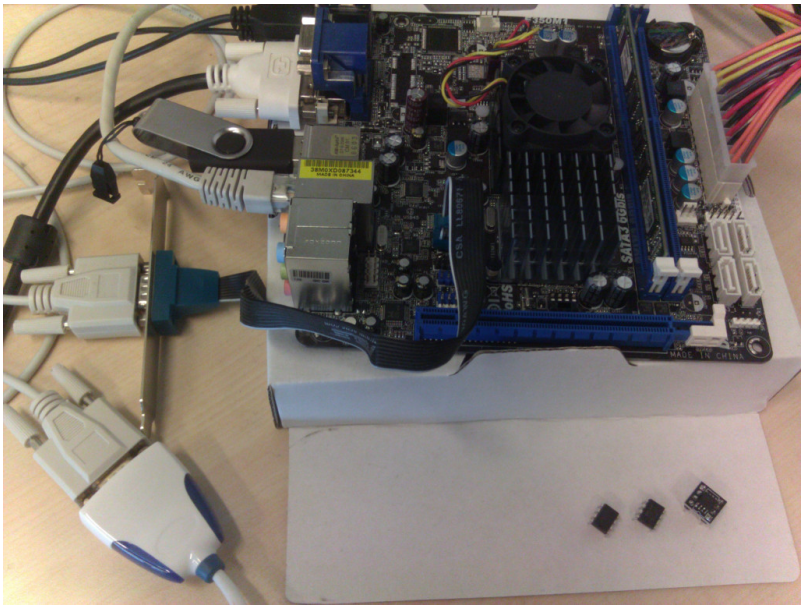

Fig. 3: Photograph of the E350M1 motherboard.

3 Registers

There are indications that both Intel and AMD use SRAM to build the register banks present in their CPUs [13], although this is not explicitly mentioned in the specification charts for their CPUs. The register banks contain, among others, general-purpose registers, MMX vector registers, and XMM vector registers. Of these, the general-purpose registers are likely to be heavily used from the moment of system start, since many of them are required to be used in basic instructions. The XMM registers, however, can only be accessed by the use of the Streaming SIMD Extensions (SSE) instruction set, which is unlikely to be used by the system startup code. They are therefore good candidates to check for PUF behavior.

However, the *AMD64 Architecture Programmer's Manual Volume 2: System Programming* [14] contains several statements which give reason to believe that it would be extremely hard, if not outright impossible, to get to the power-on state of the register banks. For instance, Table 14-1 of that document shows the initial processor state that follows `RESET` or `INIT`. The table lists a deterministic state for all the general-purpose registers, most of which get initialized to 0. The 64-bit media state (MMX registers) and the SSE state (XMM registers) are also initialized to 0 after `RESET`. After `INIT`, however, they are apparently not modified, but since it is not possible to initialize a processor without going through power-on `RESET` at the beginning, this does not help either. Volume 1 of the *Programmer's Manual* also states that, upon power-on, all YMM/XMM registers are cleared. This confirms the conclusions drawn from the table in Volume 2.

Experimental results show that the register banks are indeed not usable as PUFs on our testing machines. To explain this conclusion, we will describe the x86/AM64 boot process, and discuss how to dump the state of the XMM registers during different stages of the boot procedure.

3.1 Boot process

The boot process for an AMD64-based machine consists of several steps. The Southbridge loads the initial firmware code (BIOS or UEFI), and the processor starts executing from the `RESET` vector (address `0xFFFFFFF0`). This code performs CPU initialization and initialization of other mainboard components like the Super-IO chip, responsible for input-output through devices such as the serial port, and the memory controller, responsible for driving and communicating with main memory. Next, it searches for all bootable devices and finally loads the bootloader from the desired location.

The bootloader allows the user to select between different operating systems, loads the desired operating-system kernel and any other required resources, and then hands over control to this kernel. From this moment on the operating system is in control.

One of the main differences between BIOS and UEFI boot options is that a BIOS system will, in order to start the bootloader, drop the CPU back into 16-bit real mode, whereas a UEFI system can directly load the bootloader in 32-bit protected or 64-bit long mode. We have looked at systems using the BIOS model, but our findings apply to the UEFI model as well since the UEFI model is not different from the BIOS model in how it initializes the CPU, Super-I/O, and memory controller. For the rest of this paper, when discussing bootloader and boot firmware, we assume the BIOS model.

This division of stages in the boot process is also reflected in the complexity of the software running in each stage. The BIOS is small, very specialized, and designed to work for specific hardware. The bootloader, in turn, is somewhat larger, somewhat more portable, but still has a very limited set of tasks. Finally, an operating-system kernel is often large and complex, and designed to deal with many different hardware configurations and many different use cases. If PUF behavior can easily be exposed at the operating system level, without edits to the underlying layers, this enables wide deployment with relatively little development. If, however, the BIOS needs to be edited, then deploying a system using these PUF results would require edits to each mainboard that the system will use. The tradeoff here is that a solution which does not require edits to the BIOS and bootloader would implicitly trust these components, whereas a solution where the BIOS needs to be edited would be able to work with a much smaller trusted base system.

Because of these considerations, we decided to explore all three options. In the following sections, we first look at the kernel level, before going to the bootloader, and finally to the BIOS.

3.2 Kernel

The operating-system kernel is started by a bootloader in our test setup. We can only be sure to read potentially uninitialized values from registers if we read the state of the registers as early as possible, before they are used either by the operating system or by user processes. Thus, the register state must be stored during the startup-process of the operating system. This requires us to modify the source code of the operating-system kernel. Therefore, the obvious choice is to use an open-source kernel. We decided to use Linux.

Our code that reads out and displays the contents of the XMM registers consists of two parts: a kernel patch that stores the content of the XMM registers right after those registers have been made available and a kernel module that gives access to the stored data after the boot process has been finished.

Kernel patch. Before XMM registers can be accessed, the processor must be switched to the correct mode using the `CR0` and `CR4` control registers [14, Page 433]. This happens in the function `fpu_init` in file `arch/x86/kernel/i387.c` of the Linux kernel. Before this function is called, the kernel does not have access to the XMM registers. Thus, it is not possible that the XMM registers have been used before within the kernel and that potential PUF data in those registers has been overwritten by the kernel.

We are storing the data of all XMM registers into memory right after the control registers have been set, in order to ensure that our code is the first kernel code that accesses the registers. We use the instruction `FXSAVE` in order to save all the FPU and XMM registers to memory at once; the kernel patch adds only 5 lines of source code.

Kernel module. Displaying or permanently storing data in the very early phase of the kernel boot process is tedious. Therefore, we simply store the data at boot time and make it available to user space applications once the boot process is finished via a kernel module. The kernel module provides entries (one for each CPU core) in the `proc` file system that can simply be read in order to obtain and display the XMM register data.

Results. We tested our code on two AMD64-based machines, first on a surplus office machine with an AMD Athlon 64 X2 3800. Later, we re-ran the tests on the dedicated test-board with an AMD E350 CPU described in Section 2. Both CPUs are dual-core CPUs. On both boards, all XMM registers on the second CPU core contained all 0. The registers on the first CPU core contained some data, some of it stable over several reboots, some of it varying. However, some of the registers obviously contained ASCII code, e.g., the strings "`GNU core`", "`GB.UTF-8`", and "`: <%s>`". This indicates that the XMM registers have been used by the boatloader — if not directly in the source code then maybe by C standard-library calls like `memcpy`, `memcmp`, or string operations; disassembling the GRUB boatloader shows many occurrences of vector instructions on XMM registers.

Thus, at the time of kernel startup, the initial status of the registers has been modified and they cannot be used as PUF. Therefore, in the next step we investigated the status of the XMM registers before the kernel is started, i.e., in the early stages of the bootloader.

3.3 GRUB

The bootloader is a user-controlled piece of software, often installed into the boot sector of one of the hard disk drives. However, it runs still fairly early in the boot process. This combination of factors makes it a good candidate for attempting to find uninitialized SRAM in the XMM registers of a CPU.

GRUB patch. GRUB (GRand Unified Bootloader) is a free open-source bootloader for AMD64 systems [15]. It is one of the most popular bootloaders used to boot Linux systems and fairly easy to modify. After GRUB starts, it switches the CPU back into 32-bit protected mode as soon as possible. Then it does some more machine initialization and checks, during which it initializes the terminal console, either over the VGA output or serial output. Next, it loads all the modules it requires, loads its configuration, and displays the boot menu for the user to select an operating system.

In the previous section, we mentioned that disassembly of GRUB shows many uses of the XMM registers. However, at the moment when GRUB starts, the CPU is still in 16-bit real mode. Therefore no XMM registers are available to be used. In order to be early enough to read uninitialized registers, we changed the GRUB source code so that immediately after machine and terminal initialization, we enable access to the XMM registers ourselves, then read the register contents of the XMM registers `XMM0` to `XMM7`. Next, we write them to the terminal. First we allocate a block of memory with a size of 1024 bits (128 bits for each register) and fill it with a known pattern. Next, we enable SSE-instructions on the CPU in the first `asm`-block. Immediately after that we copy the contents of each register to the memory region allocated before, in the second `asm`-block. We do not use the `FXSAVE` instructions here, rather, we perform a single `MOVUPD` instruction for each register we want to store. Finally, we write the values from memory to the console. Disassembly of the resulting GRUB image shows that, indeed, our reading of the XMM registers is the first use of these registers within GRUB.

Results. Again, we tested our code on the surplus office machine described above and later also on the dedicated test mainboard. Unfortunately, on the first test-machine the contents of all registers except for `XMM0` were 0. `XMM0` was filled with a static value which turned out to be a fill-pattern used in the initialization code of main memory in AMD-supplied BIOS code. These values were stable over repeated tests. This indicates that at this point the registers have been zeroed and that at least register `XMM0` has been used already by the BIOS. For the same reasons as before, this means that at this point the XMM registers cannot be used as PUF, neither for randomness nor for fingerprinting. Therefore, as the next step we turned to the BIOS in the attempt to read data usable as a PUF from the registers.

3.4 Coreboot

As stated before, the BIOS is the first code run by the CPU. It detects and initializes the hardware and firmware, puts the CPU in the correct mode, runs software that makes it possible to configure the BIOS itself, and loads and runs the bootloader. The BIOS is the earliest step in the boot process that can be controlled, unless one has access to the CPU microcode.

The BIOS is loaded from an NVRAM chip. Often, its machine code is readable by reading out the NVRAM chip or by dumping the contents of BIOS updates. However, it is not easy to edit the BIOS code without access to its source code, which most mainboard vendors do not provide. Luckily, it is not necessary to reverse-engineer the closed-source BIOS provided by the mainboard vendors; there is an alternative: coreboot, formerly linuxBIOS, is a free open-source machine-initialization system [12]. It is modularly built so that it can function as a BIOS, a UEFI system, or in several other possible configurations.

Mainboard selection. Coreboot, despite its modularity, needs to be ported to every individual new mainboard for which support is desired. This is caused by subtle differences in hardware configuration, and is even required if a board uses chips which are all already supported by coreboot. Instead of porting coreboot to the AMD Athlon 64 X2 3800 mainboard mentioned before that we already had "in stock", we decided to acquire a board that coreboot had already been ported to by the community; our first requirement for the board was that it must support modern AMD64 CPUs.

Since the BIOS resides in an NVRAM chip on the mainboard, the only way to install a new BIOS is by flashing this chip. Most modern mainboards have this flash-capability built into the mainboard itself and software running in the operating system can flash the BIOS in order to enable user-friendly BIOS updates. However, should a modification to the BIOS source code render the system unbootable, this on-board capability will obviously not be available. Therefore an additional requirement was that the mainboard that we were going to use must have a socketed NVRAM chip rather than one soldered onto the board. This would allow us to boot the board with a "good" chip, then switching the chips and re-flashing the bad one.

Because of these requirements, our choice was the ASRock E350M1 mainboard described in Section 2.

Coreboot patch. The coreboot boot process begins the same as described in Section 3.1: the Southbridge loads the coreboot image, then the CPU starts processing from the `RESET` vector. The first thing coreboot does is to put the CPU into 32-bit protected mode. It then does some additional CPU initialization, initializes the level-2 cache as RAM for stack-based computing, initializes the Super-IO chip for serial port output, and then starts outputting diagnostic and boot progress information over the serial port. It initializes the memory controller, and eventually it loads the payloads stored in NVRAM, which can vary: a VGA ROM to enable VGA output, a BIOS or UEFI implementation, an operating-system kernel directly, or several other possibilities.

As soon as the cache-as-RAM initialization is done, memory is available to store the values of the XMM registers. We changed coreboot similar to how we changed GRUB. First, we allocate a buffer of 1024 bits of memory and fill them with a known pattern. Then we copy the contents of the XMM registers to the buffer. At this point, there is no interface initialized to send data out of the CPU, except for a very rudimentary POST code interface which can send one byte at a time and requires a special PCI card to read it. This is inconvenient at best, so we allow coreboot to continue machine initialization until the serial port is enabled. Then, we write the values previously read from the registers out over the serial console.

Results. This time, all the registers contain 0 on our test machine. Manual analysis of a disassembly of the coreboot firmware image flashed to the device shows that `XMM0` and `XMM1` are at some earlier point used to temporarily store data, but `XMM2`–`XMM7` are not used before being copied by the modified code. This implies that the documentation is correct, and there is no way to get access to uninitialized SRAM state by using XMM registers.

4 Cache

The AMD64 architecture defines the possibility of several levels of cache, while leaving the exact implementation to manufacturers of actual CPUs. As mentioned before, caches are usually implemented as SRAM. Therefore, reading the bootup-state of cache could be another source of PUF behavior.

4.1 Cache operation

During normal operation of an AMD64-based machine, main memory is available through a memory controller. The use of caches speeds up memory accesses by granting the CPU fast read and write access to recently touched data which would otherwise have to be fetched from main memory. On the AMD64 architecture, the data stored in caches is always the result of a read from main memory or a write to main memory; caches act as a fast temporary buffer. It is not possible for software to explicitly write to, or read from, cache. If software needs to use data from a certain address in main memory, the corresponding cache line is first loaded into cache, then accessed and potentially modified by the software, and eventually modifications may be written back to main memory. Thus, the cache contains a copy of the data that should be in main memory, but that might not be the exact same data as what *is* in main memory because the writeback has not happened yet. When exactly reads from and writes to main memory are performed, depends on the *memory type* assigned to the section of main memory being handled. For the purposes of this paper, we will only examine the memory type *writeback* [14, Page 173].

On multicore systems and cache-coherent multi-socket systems, another problem is that the data in cache itself might not be the most up-to-date copy of the data. Because of this, the cache controller must keep track of which data is stored in which location (a specific cache or in main memory) at what time. In order to keep track of this, the MOESI protocol is used that allows cache lines to be in one of five different states: *Modified*, *Owned*, *Exclusive*, *Shared*, and *Invalid* [14, Pages 169–176].

Many modern AMD64 CPUs support what is known as cache-as-RAM operation. This uses the level-2 cache in each CPU core to enable stack-based computing during the early boot process. At this point the memory controller has not yet been initialized, so main memory is unavailable [16, Pages 32–33]. In cache-as-RAM operation mode, the memory state *writeback* is assigned to all available memory addresses. After the CPU received a `RESET` signal, the entire cache is in the state *Invalid*. In writeback mode Invalid state, any memory read will trigger a "read miss", which would normally cause a read from memory into cache, and put the cache line in either *Shared* or *Exclusive* state. Any memory write will cause a "write miss", since the line needs to be modified and held as Modified in cache. Therefore, a write miss would normally cause a read from memory, modify the corresponding data, and put the cache line in *Modified* state [14, Pages 169–171]. However, the documentation does not state what happens when these misses are encountered during the early boot process when the memory controller is still disabled. It could be the case that any read from main memory will be handled within the CPU to return some static value, e.g., zero. It could also be the case that the cache is not actually modified on a read, in which case reading a block of memory might give us the power-on state of the SRAM cells in the cache.

4.2 Coreboot

The cache-as-RAM initialization code used by coreboot, written by AMD, contains instructions to explicitly zero out the cache area used as stack. Furthermore, a comment on lines 51–58 of `src/cpu/x86/16bit/entry16.inc` (one of the source files used to define the earliest stages of the coreboot boot process before the CPU is switched to 32-bit protected mode) implies that coreboot used to explicitly invalidate the cache at that point, but no longer does for performance reasons. This could imply that power-on values from the cache are indeed readable after cache-as-RAM initialization, if the instructions to explicitly zero the cache are removed.

Coreboot patch. To test this, we replaced the instructions zeroing out the cache with instructions filling it with a known pattern. Then we allowed the boot process to continue until initialization of the serial console. As soon as the serial console was available, we output the entire contents of the memory region used as stack, and confirmed that the known pattern was there. This ensures that we were modifying the correct code, and that the values were not being changed between the initialization of the cache and the output. After this test, we simply removed the instructions writing the pattern entirely to get the power-on state of the SRAM. These patches to coreboot should be applied separately from the earlier, register-related patches.

Results. Unfortunately, like in the previous experiments, the output consisted mostly of zeroes, and the parts that were non-zero were clearly deterministic and at the top of the memory region. This part of the memory most likely is the region of the stack that already has been used by function calls before and during serial console initialization. Therefore, also cache-as-RAM does not provide access to SRAM in bootup state; the CPU transparently takes care of wiping the cache before the first read access.

5 Discussion

Although we did not find a way to access and read either registers or caches before they are initialized, technically it would be possible to use them as SRAM PUFs. Thus, CPU vendors could enable these hardware features for the use as PUFs probably with relatively small modifications to their chip designs. However, if registers and caches will be made available to be read by software (either within the BIOS code or at bootloader/kernel level), they would not be protected against an attacker with physical access to the machine. In case the attacker is able to read the PUF, he would be able to reproduce the fingerprint and to impersonate the machine. In case the attacker is able to deploy malware in the early boot process, he would be able to manipulate the PUF state and thus he could influence, e.g., random number generation based on the PUF. Therefore, if CPU vendors decide to provide access to uninitialized SRAM state, further protection of PUF data is required. An instruction-set extension as proposed in [11], where the PUF data never leaves the CPU, seems to be the best way to implement this.

We have shown that the embedded SRAM in AMD64 CPUs, at least for the model we tested, is indeed not usable as a PUF. For this, we have made modifications to several open-source software packages. We release these modifications into the public domain; they are available online.

References

[1] B. Gassend, D. Clarke, M. van Dijk Srinivas, and Devadas. “Silicon physical random functions”. In: *ACM CCS ’02*. ACM, 2002, pp. 148–160.

[2] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. “A technique to build a secret key in integrated circuits for identification and authentication applications”. In: *IEEE Symposium on VLSI Circuits 2004*. IEEE, 2004, pp. 176–179.

[3] D. Suzuki and K. Shimizu. “The Glitch PUF: A New Delay-PUF Architecture Exploiting Glitch Shapes”. In: *CHES ’10*. Vol. 6225. LNCS. Springer-Verlag, 2010, pp. 366–382.

[4] R. Maes, P. Tuyls, and I. Verbauwhede. “Intrinsic PUFs from Flip-flops on Reconfigurable Devices”. In: *WISSec ’08*. https://www.cosic.esat.kuleuven.be/publications/article-1173.pdf. 2008.

[5] Y. Su, J. Holleman, and B. P. Otis. “A Digital 1.6 pJ/bit Chip Identification Circuit Using Process Variations”. In: *IEEE JSSC* 43.1 (2008), pp. 69–77.

[6] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. “FPGA Intrinsic PUFs and Their Use for IP Protection”. In: *CHES ’07*. Vol. 4727. LNCS. Vienna, Austria: Springer-Verlag, 2007, pp. 63–80. ISBN: 978-3-540-74734-5. DOI: http://dx.doi.org/10.1007/978-3-540-74735-2_5.

[7] R. van den Berg, B. Škorić, and V. van der Leest. "Bias-based modeling and entropy analysis of PUFs". In: *Proceedings of TrustED '13*. TrustED '13. ACM, 2013, pp. 13–20.

[8] A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede. "Secure PRNG Seeding on Commercial Off-the-shelf Microcontrollers". In: *Proceedings of TrustED '13*. TrustED '13. ACM, 2013, pp. 55–64.

[9] A. Schaller, T. Arul, V. van der Leest, and S. Katzenbeisser. "Lightweight Anti-counterfeiting Solution for Low-End Commodity Hardware Using Inherent PUFs". In: *TRUST '14*. Vol. 8564. LNCS. Springer-Verlag, 2014, pp. 83–100.

[10] D. J. Bernstein, T. Lange, A. Schaller, P. Simons, and A. van Herrewege. *PUFFIN – Physically unclonable functions found in standard PC components: Scientific contribution of WP1*. http://puffin.eu.org/D1.1-results-WP1.pdf. 2013.

[11] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. "OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms". In: *ACM CCS '13*. ACM, 2013, pp. 13–24.

[12] *coreboot*. http://www.coreboot.org/Welcome_to_coreboot (accessed 2014-07-25). 2014.

[13] M. Bohr. *22nm SRAM announcement*. http://download.intel.com/pressroom/kits/events/idffall_2009/pdfs/IDF_MBohr_Briefing.pdf. 2009.

[14] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. 3.23. AMD. May 2013.

[15] *GNU GRUB*. https://www.gnu.org/software/grub/ (accessed 2014-07-25). 2014.

[16] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 14h Models 00h-0Fh Processors*. 3.13. AMD. Feb. 2012.