MASTER

Professionalizing hardware-based memory acquisition for incident response scenarios cold boot using Coreboot and the Intel memory scrambler

van Heijningen, N.

Award date:
2017

PROFESSIONALIZING HARDWARE-BASED MEMORY ACQUISITION FOR INCIDENT RESPONSE SCENARIOS

COLD BOOT USING COREBOOT AND THE INTEL MEMORY SCRAMBLER

Master's Thesis

Author:
Nico van Heijningen
0893377

Supervisors:
Lejla Batina
Boris Škorić
Ruud Schramp

Eindhoven University of Technology
March 27, 2017

ABSTRACT

Volatile memory may contain many traces important in a forensic investigation. When the odds are stacked against a forensic investigator he may have to resort to innovative memory acquisition methods. In this work we present an evaluation of the forensic soundness of different memory acquisition methods based on their applicability in a near worst-case scenario. Furthermore, we discuss the details of an acquisition method we developed based on Coreboot and the cold boot attack. It is shown that the method developed is correct to a far higher degree than related methods, and that the method's integrity is directly related to the susceptibility of specific memory modules to the remanence effect. The observation that the cold boot attack is not as consistent as anticipated has been an unexpected result. Finally, research into the Intel memory scrambler of the Ivy Bridge microarchitecture is put forth. We show that it is feasible to reproduce the working of the memory scrambler based on only 1026 bits of data and provide a detailed analysis of its working.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

ASCII American Standard Code for Information Interchange

BAR Base Address Register

BIOS Basic I/O System

BIST Built-In Self Test

BM Berlekamp-Massey

CAR Cache-as-RAM

CDMA Code Division Multiple Access

CPU Central Processing Unit

DDR	Double Data Rate
DIMM	Dual In-line Memory Module
DMA	Direct Memory Access
ECC	Error-Correcting Code
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GPT	GUID Partition Table
GUID	Globally Unique Identifier
I/O	Input/Output
ME	Management Engine
IOMMU	I/O Memory Management Unit
JEDEC	Joint Electron Device Engineering Council
LFSR	Linear-Feedback Shift Register
MBR	Master Boot Record
MCH	Memory Controller Hub
MMIO	Memory Mapped I/O
MMU	Memory Management Unit
MTRR	Memory Type Range Register
NDA	Non-Disclosure Agreement
NFI	Netherlands Forensic Institute
OS	Operating System
PAE	Physical Address Extension
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect express
PCI	Peripheral Component Interconnect
PCMCIA	Personal Computer Memory Card International Association
PIO	Programmed I/O
POST	Power-On Self-Test

PRBS	Pseudo-Random Binary Sequence
RAM	Random-Access Memory
SATA	Serial Advanced Technology Attachment
SDRAM	Synchronous Dynamic Random-Access Memory
SMM	System Management Mode
SODIMM	Small Outline DIMM
SPD	Serial Presence Detect
SSD	Solid-State Drive
TCB	Trusted Computing Base
TLB	Translation Lookaside Buffer
TOLUD	Top of Low Usable DRAM
UEFI	Unified Extensible Firmware Interface
USB	Universal Serial Bus
VGA	Video Graphics Array

1 INTRODUCTION

Computers use memory to save their state. Different types of memory are used to save different states. Depending on the speed of state transitions, the data is either saved in the processor caches, Random-Access Memory (RAM), or on the hard drive. The ability to inspect the state of a computer is of great importance in the field of computer forensics. This state may prove to be meaningful evidence in a subsequent investigation. A complicating matter is that not all memory retains its state when power is removed, it is then said to be volatile. Contemporary RAM being the main example, it returns to a predefined ground state after power is lost. In general, the manner in which to acquire a computer's memory becomes increasingly more difficult as its physical size decreases and volatility increases. This work therefore, focuses on extraction of data from RAM while a computer is (still) switched on.

1.1 PROBLEM DESCRIPTION

How to acquire all memory of a locked, yet powered on computer from a uncooperative user? That is, a (near worst-case) incident response scenario where: no privileges are available (locked computer); the machine is fully patched (no exploits available); the risk that anti-forensic measures may be in place (full-disk encryption); and a Basic I/O System (BIOS) that clears the memory on reset. Furthermore, we assume that, physical access to the device is possible (no need for remote acquisition), yet no previous interaction has been possible (no opportunity for pre-installation).

1.2 SCOPE

To scope the project into a manageable size for the time frame set, it was chosen to focus on obtaining a flat linear copy of the physical DDR3 memory of a predefined desktop system (see Appendix A). As a consequence, this work focusses solely on Intel hardware. Furthermore, as the analysis of an obtained physical memory image allows for a whole study on itself, no other requirements on post-processing or carving logical data structures out of the obtained physical data have been defined.

1.3 SCIENTIFIC QUESTION

Taking into account the problem description, four scientific questions have been defined.

- What memory acquisition methods are available and applicable in this scenario?
- How can the most suitable method be professionalized for use in a forensics lab?
- How forensically sound is the method of choice?
- What is the effect of the memory scrambling mechanism as implemented by Intel?

1.4 CONTRIBUTION

The contribution of this thesis is threefold:

I. A review of the acquisition methods available in a near worst-case incident response scenario is presented in Part I.
II. The implementation of a novel memory acquisition method is discussed in Part II.
III. An analysis of the Intel memory scrambler is provided in Part III.

Part I

LITERATURE STUDY

2 BACKGROUND

In order to aid in the understanding of this thesis, a discussion on the underlying concepts is presented in this chapter. It should be understood that some generalization is in order, to allow for a concise, yet self-contained section. A more detailed discussion of recent Intel computer architectures may, for example, be found in [11].

2.1 COMPUTER ARCHITECTURE

In general, personal computers are built from different components, all of which need to function in harmony. A main Printed Circuit Board (PCB), named the motherboard, houses all the parts (see Figure 1). The components communicate through hardware buses, which consist of a certain number of parallel wires called traces on the PCB. On top of the buses run different signalling protocols. The main components are the: Central Processing Unit (CPU), RAM, and different Input/Output (I/O) peripherals. The latter of which are made available through a multitude of different ports. For example, Universal Serial Bus (USB) to connect general peripherals, Video Graphics Array (VGA) to connect monitors, Serial Advanced Technology Attachment (SATA) to connect hard disks, and Ethernet for networking.

The manner in which components are connected and communicate differs between hardware generations and models. In general there are two main components responsible, the northbridge and the southbridge, which together can be identified as the chipset. The northbridge or Memory Controller Hub (MCH), is connected with a fast bus to the processor, it determines where memory addresses are routed, and is responsible for fast communication between the: processor, memory and expansion buses such as Peripheral Component Interconnect express (PCIe). The southbridge is responsible for the communication between peripherals and the northbridge. Whereas older hardware used to have components physically placed centimeters apart due to their size and heat restrictions, recent developments focus on placing more components on a single chip. For example, in most modern CPUs, the communication with the main memory is now handled by logic placed on the same silicon die as the CPU cores. So the clear boundaries between the: CPU, northbridge, and southbridge have faded.

A general x86 instruction set compatible CPU consists of registers, caches, and (multiple) processing cores. The registers save the state of the processor, the caches buffer (recently requested) data from main-

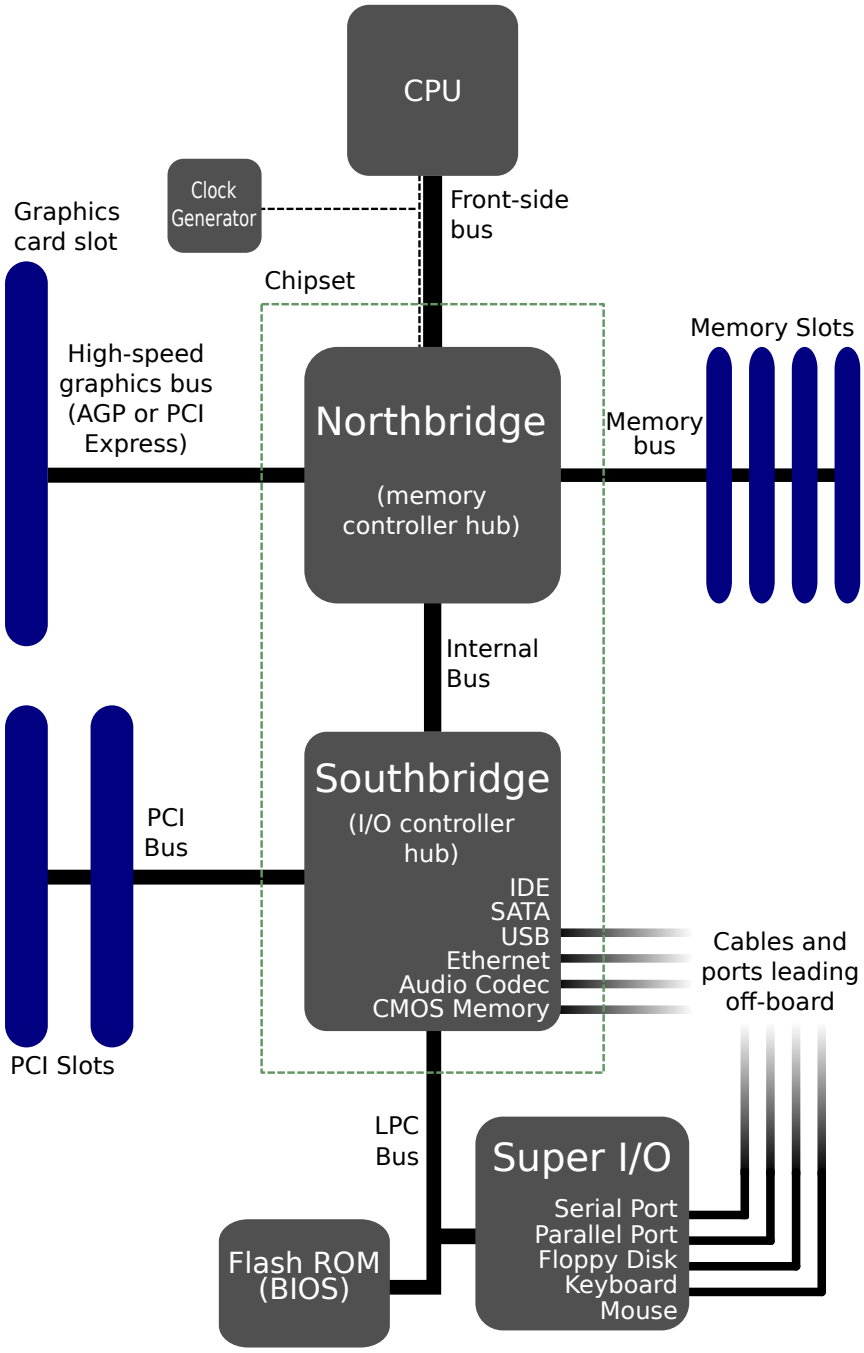


Figure 1: *Graphical representation of a motherboard. Source: Moxfyre at English Wikipedia*

memory, and the cores perform operations on the data. There may be different levels of cache with different purposes e.g. an instruction cache and a data cache. For more details on the x86 architecture we kindly refer to [31].

Contemporary main-memory or RAM is generally packaged in the form of Dual In-line Memory Modules (DIMMs), of the Synchronous Dynamic Random-Access Memory (SDRAM) type and based on one of the four Double Data Rate (DDR) standards (DDR1 through DDR4 [34, 35, 37, 38]) defined by the Joint Electron Device Engineering Council (JEDEC). DDR3, on which we focus, is discussed in more detail in Section 2.3.

2.1.1 *Boot sequence*

When a computer is started, it goes through a boot sequence, initializing all connected hardware to a working state such that an Operating System (OS) can be started. The first code a computer executes is located at the reset vector. This is a memory address which is generally routed to a non-volatile memory chip where firmware and BIOS code is saved. After executing the firmware and BIOS or Unified Extensible Firmware Interface (UEFI) code, the Master Boot Record (MBR) or GUID Partition Table (GPT) is loaded, which is usually located on a hard disk. This invokes the bootloader which loads the kernel. As soon as the kernel and other OS specific components have finished loading, user applications may be started.

2.2 MEMORY ADDRESSING

The location of data in RAM is defined by its address. Each memory address points to a single byte of data and the address bus is the physical representation of a memory address. The width of the address bus determines the length of the memory addresses and therefore the maximum number of possible addresses[1]. In turn limiting the amount of memory that can be used by a system. For example, addresses of 32 bits allows for four gigabytes of data to be located[2]. When more than four gigabytes of memory is to be used, a 32 bit wide address bus by itself is insufficient.

Over the course of time, the desire for speed and backwards compatibility has led to the introduction of different processor execution modes. Each of those modes allows a processor to address more memory. Mainly due to legacy reasons, older modes are still supported by newer CPUs. *Real mode* is the 16 bit legacy mode all x86 processors start in. Only a single megabyte of memory is addressable in this

1 Multiplexing reserved.

2 2^{32} addresses can point to 4294967296 bytes / 1024 (KiB) / 1024 (MiB) / 1024 (GiB) = 4 GiB

mode. *Protected mode* is the native operation mode of the 32 bit x86 instruction set. *Long mode* is the native operation mode of the 64 bit x86 instruction set, and allows more than four gigabyte of memory to be addressed.

Next to these legacy modes, *System Management Mode (SMM)* was introduced. SMM transparently suspends normal execution to allow for an isolated execution environment, mainly used for power management and system security by the system's firmware. Again we refer to [31, Chapter 3 - Basic execution environment] for a more detailed discussion.

As memory acquisition requires reading all of the memory physically present on the DIMMs, there are two artifacts of these (legacy) modes that have to be taken into account: memory address routing and translation.

2.2.1 *Memory address routing*

Not all memory addresses may point to data in the physical DIMMs: ranges of memory can be routed to other components. Examples of mechanisms that cause memory addresses to be routed to different components are: Memory Mapped I/O (MMIO), Memory Type Range Register (MTRR), and static routing. MMIO maps the memory of peripherals (I/O devices) into the computer's address space. The use of MMIO allows the memory of e.g. a network card to be accessed through regular memory operations. To enable MMIO the BIOS configures the processor's Top of Low Usable DRAM (TOLUD) register with a memory address above which all memory accesses (up to the 4 GiB limit) are reserved for MMIO. The Base Address Registers (BARs) of e.g. PCI peripherals can be programmed such that those devices may respond to reads and writes to data in the MMIO range. The MMIO area is thus not backed by actual RAM. MTRRs control how ranges of memory addresses are cached. For example, they determine when an update of the cache is written back to RAM. MTRRs are used to enable Cache-as-RAM (CAR) (discussed in Section 5.3) and are discussed in more detail in [11]. MTRRs may hence cause certain reads from memory to be read from the cache only, instead of the actual RAM. Finally, the memory addresses near the reset vector may be statically routed to the BIOS chip, as in that point of the boot sequence, other hardware has yet to be initialized.

Furthermore, sequential memory accesses may be distributed over different components through the routing mechanism. For example, CPUs may include options to interleave memory accesses on the granularity most suitable for performance increase [56] and [28, 2.1.3.2.1 Dual-Channel Symmetric Mode]. By routing, for example, sequential memory addresses to different DIMMs these accesses can be pipelined and thus retrieved concurrently. Figure 2 shows an exam-

ple of the routing behavior of a computer with two DIMMs installed. It highlights the difference in the resulting linear copy of memory, between the case when no interleaving is applied and when the memory addresses are distributed in an interleaving fashion across the two DIMMs (on a RAM chip granularity). The address at which a specific region of memory starts can be altered by setting registers in the MCH. Many different regions of memory have been ascribed a specific purpose. For example, the legacy region of Figure 2 includes many smaller regions with each a different purpose. To prevent a lengthy discussion of the historical reason of existence of every single memory region, we kindly refer to the datasheets of the hardware ([28] in our case).

One important aspect of different memory regions in the context of memory acquisition are the so-called 'stolen' memory regions. These regions of memory are used by components to save their state. Examples of components using stolen regions are: the integrated graphics device, SMM, and the Intel Management Engine (ME) (for details on the latter we refer to Appendix D). Instead of using their own internal memory, the RAM is used for this. However, the regions are allocated, to the components, very early in the boot sequence, and may be routed to allow the components exclusive access. This might be the cause of serious concerns when acquiring memory; a point which we will return to in Section 6.3.

2.2.2 *Memory address translation*

A memory address may, over the course of time, translate to different locations in physical RAM. The main mechanism that introduces such translation is called paging. Next to providing isolation between code and data, paging can also be used to provide virtual memory. Virtual memory allows a computer to use more memory than actually exists, by saving lesser used parts of memory to slower memory (e.g. the hard disk). Paging introduces the distinction between logical and physical addresses. Logical (virtual) addresses are used by the code whereas the physical addresses are the locations the data is actually stored. When a logical address is accessed that does not actually reside in RAM, the OS first puts the address's data in memory before continuing execution of the program. Because the logical address space is far larger than the physical address space, over time the same logical address may point to different physical addresses.

2.2.3 *The overloading of address terminology*

The consequence of the above sections memory address routing and translation is that some literature attributes different meanings to the term 'physical memory address'. Overloading the term by using it

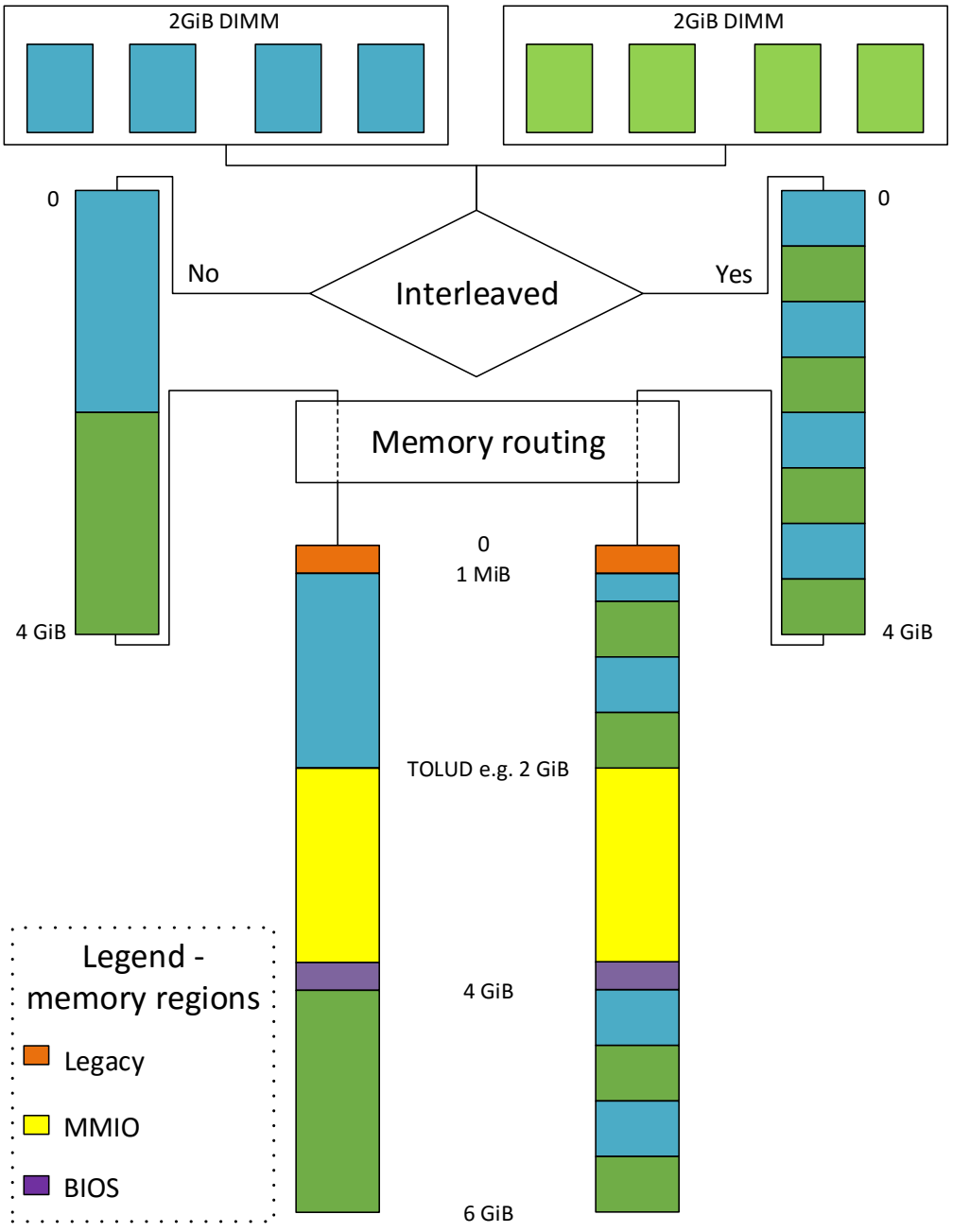


Figure 2: *Memory layout of a typical computer with two DIMMs installed. Memory routing is applied irrespective of interleaving being applied. Due to reasons of visibility, not all regions use the same scale.*

both in the context within the host and in the context of communication over the memory bus (between the host and the DIMM).

On the host side, the logical addresses are used to address virtual memory, and are translated by the Memory Management Unit (MMU) –using the paging mechanism– to physical host addresses. The Translation Lookaside Buffer (TLB) is used to cache a number of these translations to increase the mechanism's speed. The physical host addresses are next converted by the MCH to physical DIMM addresses, for example, DDR3 commands and send over the physical memory bus. When not obvious from the context we will prepend the term 'physical address' with either host or DIMM to remove ambiguity.

2.3 SDRAM AND DDR3

There are many different types and generations of volatile memory. This work focusses solely on DDR3 Synchronous Dynamic Random-Access Memory (SDRAM). The 'synchronous' in SDRAM means that all memory works based on a single clock, whereas 'dynamic' stands for the fact that the memory needs to be refreshed (the cause for its volatile property). SDRAM needs to be refreshed, as each memory cell consists of a capacitor and transistor. The transistor controls whether power can flow to/from the capacitor and the capacitor holds the memory cell's state. The cell's state is determined by the difference between a threshold value and the charge saved in the capacitor. Even when the transistor prevents charge to flow to/from the capacitor, the charge still gradually leaks to/from the capacitor. Details on the underlying physical properties regarding this leakage can, for example, be found in [21]. Once a memory cell's state can no longer be determined it is said to be in it ground state. The cell's ground state may either be a zero or one depending on whether the memory cell's capacitor is wired to power or ground. To prevent this, all capacitors need to be recharged before any of the cells' states can no longer be determined correctly. The leaking of the charge of a capacitor is called decay, and its counterpart is called remanence. The time a cell's state can correctly be determined without a refresh is named its remanence time or period. The remanence period is lower bounded by the DDR3 specification [37], defining that a refresh command is to be sent every 7.8 µs. Depending on the memory's density, typically every capacitor is required to have a minimum remanence period of 64 ms. Figure 3 shows cross sections of two different types of capacitors present inside SDRAM chips. Trench capacitors are an older technology which relatively take up more space on the silicon than stacked capacitors. The chronological development of SDRAM capacitors is discussed in [66] and an analysis of the capacitors used in contemporary SDRAM chips are provided in [40].

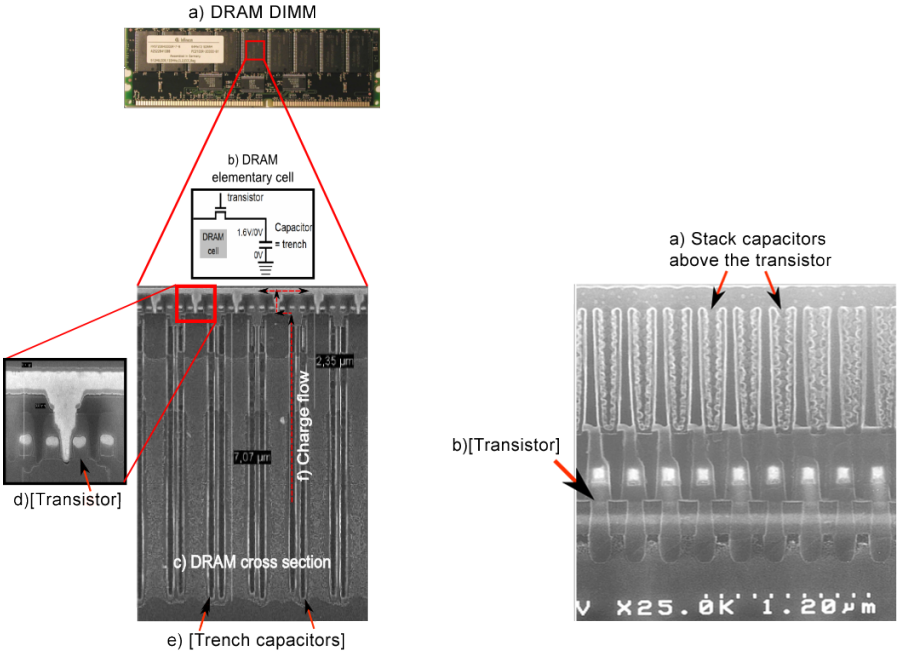


Figure 3: *Different physical appearances of capacitors. Trench capacitors are shown left (planar view), stacked capacitors are shown right (cross-section). Source:* `http://www.sdram-technology.info/sdram-cross-section.html`

The DDR3 specification [37] supports a plethora of different variants and extensions, providing SDRAM manufactures the room to innovate. Innovations are typically related to advances in integrity, speed, and power consumption. Due to reasons of conciseness and relevance, we refrain from providing a complete overview of all options and their details. However, many of these advances have been standardized by JEDEC and some of the different options that may be encountered are: speed (frequencies, timings); integrity (registered (RDIMM), fully buffered (FBDIMM), load reduced (LRDIMM), Error-Correcting Code (ECC)); power usage (DDR3L (low voltage), DDR3U (ultra low voltage)); and form factor (Dual In-line Memory Module (DIMM), SODIMM (small outline), LPDIMM (low profile), VLPDIMM (very low profile)). Each memory module is labeled, the contents of this label is typically standardized by its manufacturer[3]. Most of the aforementioned options should be recognizable on this label. Furthermore, every module contains a small amount of persistent memory, used to save Serial Presence Detect (SPD) information, again standardized by JEDEC [39]. The SPD data includes information regarding the type of DIMM in an electronic format. SPD is used by a computer's memory controller to enable automatic initialization of different types of DIMMs. Not every memory controller may sup-

3 See for example [42, 49, 60], a preliminary JEDEC specification has also been released but has, to the best of our knowledge, never been standardized [36].

port all types of memory modules. Typically a memory controller only supports one generation of DDR and only higher-end motherboards support integrity enhancing options. A memory controller initializes and communicates with a DIMM by transitioning through the DIMM's internal state machine as defined in the DDR3 standard [37, 3.1 Simplified State Diagram].

The details of SDRAM discussed above lie at the basis of the different factors that influence the cold boot method; as will be discussed in more detail in Section 3.6 and Section 7.3.

3

RELATED WORK

A body of research has been developed on the topic of memory acquisition. Due to some subtle, yet important differences in scope and scenario, not all work fits that of ours. Therefore, different methods of memory acquisition are reviewed; promising methods are presented in more detail. Furthermore, a justification of each method's suitability in our setting is provided. Every justification is accompanied by an evaluation of the method's forensic soundness, based on the following criteria.

3.1 METHODOLOGY FOR EVALUATING FORENSIC SOUNDNESS

The quality of an acquisition method and the data it produces is of great importance in a forensic context. Vömel and Freiling have distilled this idea of forensic soundness (of acquired volatile memory) into three criteria: correctness, atomicity, and integrity [70, 71]. Vömel and Freiling provide a formal definition of the criteria, together with a justification of their independence and claim that satisfying these three criteria implies a forensically sound copy of memory; here we only provide an intuitive description. The acquired memory is deemed forensically sound if it is:

- an exact copy of the data in memory i.e. no data skipped/missed (*correctness*).
- free of any concurrent system activity i.e. other processes are prevented to write to memory (*atomicity*).
- the memory is not modified after a certain point in time i.e. the start of acquisition (*integrity*).

The criteria have been applied to specific implementations by Vömel and Stüttgen [72] and Gruhn [15, Chapter 4], as a whitebox and blackbox methodology respectively. Vömel and Stüttgen restricted themselves to software only acquisition methods, and Gruhn has limited his scope by taking correctness for granted. Their work has shown that most memory acquisition methods are only able to fulfill the three criteria up to a certain degree. We have based our evaluation on their findings, but differ our methodology on two points. First, instead of checking the correctness of a region of memory, we explicitly evaluate a method's correctness based only on the question whether it can acquire all memory available. Second, instead of evaluating specific implementations, we aim to review the general acquisition methods in relation to our scenario as defined in Section 1.1; to be able to do this we introduce the following generalization.

Although not explicitly noted by Vömel and Freiling, we argue that their criteria encompass the concept of anti-forensics. Anti-forensics is defined as: "any measure that prevents a forensic analysis or reduces its quality" [15] and therefore the goal of any such measure is to reduce the degree of correctness, atomicity, and integrity. One may argue that the a-priori chance of the presence of anti-forensic measures may be small, based on a risk assessment on the background of the user. However, such an assessment may prove to be difficult to perform or its result to be biased, due to the possible lack of information on a user. We argue that chance of the success of anti-forensic measures is proportional to the size of the Trusted Computing Base (TCB) of the acquisition method and that therefore a method requiring a small TCB is preferred above a method requiring a large TCB.

3.2 SOFTWARE

A multitude of applications have been developed over the years with the aim to secure volatile memory. For all major recent OSes a program can be installed and run to acquire memory. Due to its ease of use and low impact on the stability of the system, we consider software to be a preferred memory acquisition method.

However, the fact that the software runs on the same machine from which the memory needs to be acquired, has two implications. First, the acquisition software alters the state of the machine, as it loads itself in memory, overwriting the data to be acquired. Second, software has the largest TCB possible. The first argument may –for a large part– be overcome by using software of a very small size, only overwriting specific regions of memory. Furthermore, the software is permitted to use portions of memory it has already secured, allowing additional functionality to be introduced over time. Different stages and compression techniques can, therefore, be considered [4]. As for the second argument, not only may the integrity of the TCB, including the OS, pose difficult to predict beforehand, it may also prove problematic to determine a-posteriori.

Finally, as user-land programs are not allowed to read all (kernel) memory, the software needs to be installed with root privileges. Excluding a software-only approach in our scenario.

Evaluation: not applicable in our scenario. Although existing software scores high on the criteria of correctness, it scores relatively low on the criteria of atomicity and integrity [70, Tables 4.2, 4.3, 4.4].

3.3 DIRECT MEMORY ACCESS (DMA)

DMA allows components, other than the CPU, direct access to a computer's RAM and is used to speed up memory accesses by peripherals. Different external physical ports allow for DMA access. Exam-

ples are: PCI/PCIe [10, 14, 75]; PCMCIA, ExpressCard, PCCard [75]; FireWire [5, 12]; and Thunderbolt [62, 63]. Similar to software, the DMA method is performed on the same hardware from which the memory is acquired. Again allowing the method to be subverted by any of the components in its TCB [59]. DMA attacks may be mitigated through I/O Memory Management Unit (IOMMU) and OS protection mechanisms. The IOMMU acts as a kind of firewall, only allowing a device to read memory ranges specifically allocated to it. Recent OSes disable DMA once the device is locked[1]. Originally DMA only allowed for 32-bit addressing, therefore many DMA methods can acquire no more than four gigabytes of memory. Workarounds for this may require hardware or OS support, both of which cannot be applied in our scenario. Finally, the physical ports, e.g. Thunderbolt, may be unavailable, inaccessible or non-existent, and as some ports may not be hot-pluggable these may require pre-installation e.g. PCI/PCIe. These requirements exclude a DMA-only approach in our scenario.

Evaluation: not applicable in our scenario. Existing DMA methods score the lowest on the criteria of atomicity and integrity [15, 4.5.2.2 inception] when compared to other methods; its score on correctness is unclear.

3.4 System Management Mode (SMM)

Multiple authors [57, 73] have opted to run software in System Management Mode (SMM) to acquire memory. Although the switch to SMM would halt other software and increase the method's atomicity, to obtain code execution in SMM one is required to either: (1) run software at the most privileged level; (2) pre-install the code to the BIOS chip and invoke it through a secure channel e.g. hardware switch; (3) patch the BIOS chip, reboot the system, and start executing own code from the reset vector.

The first possibility is not an option in our case. The second option is discussed in [57, 73], yet not applicable in our case due to its pre-installation requirement. The third choice defies the use of SMM for the most part, and other than running in SMM, is the same as the method discussed in the next section.

Evaluation: the same as BIOS modification in our case.

3.5 BIOS modification

Schramp has shown that a BIOS chip can be replaced whilst the system is still running [61]. By replacing the chip with one which houses

1 "OS X Lion disables DMA when the user is logged out/screen is locked and FileVault is enabled. [...]" http://www.breaknenter.org/projects/inception/

SerialICE[2] Schramp was able to dump the computer's memory. SerialICE is software which allows one to: run a BIOS in the QEMU[3] emulator on one machine, and proxy all hardware accesses to the SerialICE mini shell running from the BIOS chip on another machine. By resetting the computer, it can be made to run the replacement BIOS's code from the reset vector. Schramp modified SerialICE such that it dumps the computer's memory directly after RAM initialization. The memory was dumped as ASCII hexadecimal Power-On Self-Test (POST) codes to a POST dongle. The POST codes are normally shown on a seven-segment display of a POST card indicating at what stage of self-test the BIOS is. The codes are typically used as extra information for troubleshooting, but allow for a slow communication channel. Gruhn [15, Section 5.5.2] attempted to reproduce the work of Schramp, yet was only partially able to do so. Gruhn was required to pre-install a BIOS chip socket such that the chip could be replaced without soldering.

Modifying the BIOS in general has been an active topic of research, often from the perspective of rootkit development [7, 8, 23]. This has led to the development of UEFI and Intel Boot Guard. The UEFI specification modernizes the older BIOS interface, and among other things, allows for cryptographic verification of firmware updates. Thereby preventing illegitimate modification of the firmware from the OS. However, implementations of the UEFI specification have proven to be vulnerable [3, 24, 26, 43]. To this end, Intel devised Boot Guard [29, 30]. Although the details on Boot Guard are scarce, the method seems to bind the integrity of contents of the BIOS chip to keys burnt into the hardware. Preventing illegitimate modification of said contents, which includes the firmware, as this check is performed on every power-on.

All in all, the suitability of this method is very platform-specific due to hardware specific initialization requirements and possible countermeasures. Therefore, this method may pose to be non-trivial to port to different platforms. Furthermore, because code execution can only be obtained through a reset (possibly depriving a computer's memory of power), the data remanence property (discussed in the next section) needs also to apply.

Evaluation: applicability conditioned on the presence of Intel Boot Guard and data remanence. Mixed results have been achieved during reproduction. We argue that BIOS modification scores high on the criteria of correctness, atomicity and integrity.

2 https://www.serialice.com
3 http://qemu.org/

3.6 COLD BOOT

The cold boot method is based on the principles of memory retention (as discussed in Section 2.3), known in theory before but made famous thanks to the seminal paper of Halderman et al. [20]. Volatile SDRAM memory retains its state for a short period of time without power, cooling the memory increases this period. There are two general variants of this method. A *first order* cold boot attack, in which the same computer is reset to start acquisition software, typically loaded from a bootable device. A *second order* cold boot attack, in which the memory modules (DIMMs) are transplanted to a different computer which runs the acquisition software. Typically software such "memimage" as released by the original authors[4] or "msramdump" by Wesley McGrew[5] is used to perform a cold boot attack. However, this software does have the same consequences described in Section 3.2; overwriting parts of the memory to be acquired. Even more so, Halderman et al. noted that: the BIOS may overwrite small parts of memory with its own code and data; destructive POST routines may destroy large chunks of memory; and ECC memory may be initialized entirely to a known initial state by the BIOS [20, section 3.4 BIOS footprints and memory wiping]. A more detailed discussion on these concerns is provided in Section 6.3.

As discussed in section 2.3, the period of time during which memory survives without power is labeled the retention period. Not all memory cells may survive over the whole retention period, some cells may decay faster than others. The length of the retention period is essential to the success of this method, yet depends on many different factors. Lindenlauf, Höfken, and Schuba [45] have identified the variables at play to be: SDRAM type; SDRAM manufacturer; individual SDRAM; cold boot mainboard; multi-channel mode; SDRAM temperature; and SDRAM time without power. Carbone, Bean, and Salois [9] have identified the physical properties underlying these factors to be related to the: memory density, residual capacitance of the cold boot motherboard, and resolution of the production process. A point we will return to in Section 7.3.

Over the years, mixed results in performing cold boot attacks have been achieved. Hannay and Woodward [22] set out to verify the work of Halderman et al., but were unable to find any remnants of data aside from data of the software used. Carbone, Bean, and Salois [9] were able find remnants of data, but only for some of the tested systems[6]. Gruhn and Müller [16] claimed that cold boot attacks on DDR2 are possible, DDR3, however, was deemed impossible to acquire using the cold boot method. Gruhn and Müller note that it is unclear

4 https://citp.princeton.edu/research/memory/code/

5 http://mcgrewsecurity.com/oldsite/projects/msramdmp.1.html

6 Although extensive details of the computer systems tested are included, it is unclear from their work exactly which memory types have been tested.

whether the immunity of DDR3 is solely due to its construction type, or also owing to effects introduced by the memory controller. In a paper written from an engineering perspective, Liu et al. [46] have tested the retention times of 248 different SDRAM chips of thirty-four different DDR3 SODIMMs of five different vendors at various temperatures[7]. Through the use of an FPGA board and different testing patterns, they claim to have observed no memory cells with a retention time smaller than 1.5 seconds at 45°C. Strengthening the conjecture of Gruhn and Müller that the memory controller introduces detrimental effects. Lindenlauf, Höfken, and Schuba [45] have performed data retention experiments on fourteen DIMMs; both DDR2 and DDR3. Whereas the DDR2 experiments succeeded on a Gigabyte GA-G41M-Combo motherboard, the DDR3 experiments did not succeed on the same board. Nevertheless, the attack did work for DDR3 on the motherboard of a Asus P53E notebook. Leading the authors to claim that "[...] many boards overwrite DDR3 SDRAM with a random bit pattern during a cold boot.". Recently Bauer, Gruhn, and Freiling [2] verified that hardware vendors (e.g. Intel) have opted to apply memory scrambling. We suspect that, this hardware based memory scrambling has most likely been the cause of problems in reproducing the work of Halderman et al. on DDR3 as experienced in [16] and [45]. Memory scrambling will be discussed in greater detail in Part III.

It is good to note that, on the one hand, this method may well be the most destructive of all methods presented here. Any mistake in cooling or timing may result in only the ground state of the DIMM being acquired, not allowing for any second try. On the other hand, we argue that this method has the smallest TCB of all and that it may be applied in certain scenarios (e.g. ours) where other methods can not.

Evaluation: applicability conditioned on the presence of the remanence effect. Mixed results have been achieved during reproduction. The cold boot method scores the highest on the criteria of atomicity and integrity [15], yet the rate of decay may directly be related to the degree of integrity that can be achieved.

3.7 LEGAL REQUIREMENTS

Owing to the fact that the acquisition method will most likely be used in a forensic context, it is of importance that data acquired is deemed to be reliable evidence in a court of law. To this end we provide an overview of the relevant jurisprudence and case law.

Typically, digital evidence of this nature is to be presented to the court by a expert-witness. In the Dutch case (which we focus on) an expert-witness may be registered in the nationwide expert-witness

7 It is unclear exactly which chips of what DIMMs have been tested. Furthermore, the vendor names have been anonymized.

register[8] meant to assure the quality of the expert. However, this is not obligatory and it is up to a judge whether he allows one to act as an expert-witness. The Dutch law states: "De deskundige brengt aan zijn opdrachtgever een met redenen omkleed verslag uit. *Hij geeft daarbij zo mogelijk aan welke methode hij heeft toegepast, in welke mate deze methode en de resultaten daarvan betrouwbaar kunnen worden geacht* en welke bekwaamheid hij heeft bij de toepassing van de methode." [74] (emphasis by the author). The emphasis translates to: the expert-witness should –if possible– note which method was used and to which extent the results of the methods can be deemed reliable. Based on case law and Dutch jurisprudence ([25], [6, pp. 50-82], [54]) the following questions –that may be asked by a judge– have been formulated[9] (translation by the author):

I. What is the profession, education, and experience of the expert?
II. Does the expertise of the expert extend to the matter at hand?
III. Which method was used by the expert?
IV. Were there other methods available? If so, then why was this method chosen?
V. What is the reliability of the used method (motivation of)?
VI. Was the expert able to professionally execute the method?

Only once a suspect challenges the findings of an expert-witness a judge is required to ask these questions. It is then up to the judge as to determine whether an additional expert is required to reach a conclusive verdict. The main problem in the scenario of volatile memory is the possible destructive application of any of the acquisition methods. In other forensic disciplines, the Netherlands Forensic Institute (NFI) has established forensics technical standards, these include procedures on the (possibly destructive) securing of biological traces e.g. blood or fingerprints. To the best of our knowledge, such standards have not been established in the digital context. We argue that the use of such a standard should also raise the confidence of a judge in a acquisition method in the digital domain, as its reliability in general will become more uniform.

To aid in this we put forth two appendices: Appendix B, in which a flowchart is presented, that may help a forensic investigator in choosing the most suitable method of memory acquisition; and Appendix C which discusses the procedure that should be followed once the acquisition presented in Part II has shown to be most suitable. We conclude that the reliability of the acquisition method of choice needs to be justified. We do this by reviewing applicable scientific literature (as provided in this chapter), and by discussing the method's implementation details and validating its forensic soundness in Part II.

8 https://www.nrgd.nl/

9 In the international context the (similar) Daubert criteria [69] have long been of the de facto standard.

3.8 CONCLUSION

In a memory acquisition scenario, there are many factors involved that may vary widely. Some may be in one's own hands, whilst others may not. In the context of the predefined scenario, every method discussed in this chapter brings its own advantages and disadvantages to the table. By opting for a combination of the BIOS modification and cold boot methods, we aim to professionalize an acquisition method that is applicable even in the most worst-case of scenarios. Our hope is to produce a forensically sound acquisition method with a small TCB.

Part II

ENGINEERING COLD BOOT USING COREBOOT

4 CONCEPT AND DESIGN

4.1 CONCEPT

The general concept is to use the cold boot method together with BIOS modification. It is known that the cold boot method does not work with certain BIOSes. Therefore, it is our goal to, on the one hand, have a BIOS that doesn't wipe or overwrite memory whilst, on the other hand, to implement additional (acquisition) functionality. This idea is not new, it has been thought of before by Appelbaum, one of the authors of the original cold boot paper:

> "If you were to implement your own custom BIOS you would be able to, for example, dump the memory with having a very minimal amount of memory being stomped on. [...] With Coreboot someone, I think his name is Uwe and maybe Peter [Stuge], I forget, they implemented a small plugin for Coreboot that actually allows you to boot a machine that has this little plugin loaded and it simply dumps the memory to the screen and you can page through it." [1] (transcription by the author).

The functionality of the Coreboot plugin is present in the "coreinfo" payload of Coreboot under the name "RAM dumper"[1]. Stuge elaborated on the idea of Appelbaum by commenting:

> "Because we're doing the RAM initialization we can stop right after RAM initialization is complete and do whatever we want, it's open-source so go ahead and have some fun. Do some special dump routines, we can even utilize another trick which is called Cache-as-RAM to not have to use any RAM at all so we can really extract every single byte of data that is in RAM." [64].

There are two main advantages of this concept over the available cold boot acquisition software such as "memimage" and "msramdump". First, Cache-as-RAM (CAR) should allow the acquisition of all memory, directly after memory initialization, without overwriting any of it. Second, the time the transplanted DIMMs are without power can be reduced by detecting the insertion of a DIMM from CAR.

To the best of our knowledge, a BIOS specifically developed for memory acquisition has not been implemented before. Therefore, we

1 https://www.coreboot.org/Coreinfo

have developed a proof-of-concept implementation, the design and details of which will be discussed next.

4.2 DESIGN

The acquisition method developed has been designed to minimize the time the DIMM is without power, and should thereby reduce the amount of decay introduced. The procedure for acquiring memory is as follows:

I. Boot the acquisition machine without a DIMM and poll for the insertion of a DIMM
II. Cool the DIMM of the target machine
III. Transplant the DIMM from the target to the acquisition machine
IV. As soon as the DIMM is inserted, the acquisition machine initializes and starts refreshing the DIMM
V. The contents of the DIMM are secured to persistent memory of the acquisition machine

During the entire procedure nothing is overwritten in RAM as all code is running from CAR and the memory is acquired as soon as memory initialization has finished.

5 COREBOOT

Coreboot[1] is free open-source software and has been described as an open-source extended firmware platform. It is an alternative to the closed-source firmware and BIOSes developed by commercial independent BIOS vendors such as: Insyde, American Megatrends Inc, and Phoenix. Coreboot performs the minimal amount of hardware initialization possible to be able to execute a payload. Different payloads can be executed after Coreboot has initialized the hardware. Examples are: SeaBIOS (x86 BIOS implementation), TianoCore (UEFI implementation), a Linux kernel, or diagnostic tools (e.g. MemTest86). Coreboot's code is the first that runs on a computer[2].

5.1 BINARY SITUATION

Firmware contains large amounts of platform-specific hardware initialization code. Much of this is only provided as binary blobs by hardware vendors. Hence, open-source initialization code can only be developed by reverse engineering such blobs. This process is very time-consuming and the documentation of such code is arcane due to the lack of datasheets, as these are only available under Non-Disclosure Agreement (NDA) or are vendor confidential. Coreboot's choice to include blobs that are not (yet) reverse engineered is one of the reasons behind the Coreboot code-fork named Libreboot[3]. Libreboot aims to be completely blob free, but is only supported on a very limited set of devices because of this constraint.

5.2 MEMORY INITIALIZATION

The memory initialization code of modern architectures is an example of functionality only provided as a binary blob, without any source code e.g. Intel's Firmware Support Package [65, Chapter 4]. Fortunately, the memory initialization code of older platforms has been reverse engineered and for these platforms the source code is available. On the one hand, this allows us to take over execution directly after RAM has been initialised. On the other hand, although the code is open source, it is by no means easily understood. It con-

1 https://www.coreboot.org/

2 Before the reset vector the processor may have executed microcode updates or other logic such as the Built-In Self Test (BIST). Most of the time, this logic cannot be altered at all, or only by the hardware vendor itself through cryptographically signed updates. Therefore, it may be assumed integer.

3 https://libreboot.org/

tains many 'magic' reads and writes from and to memory addresses and offsets thereof. Furthermore, public documentation in this topic is absent.

5.3 CACHE-AS-RAM (CAR)

Coreboot is able to run compiled C code before it has even initialized RAM. The motivation to run C code so early in the boot sequence, is based on the fact that assembly code is much more difficult to reuse and maintain across many different architectures. Originally Coreboot was only able to run C code so early in the boot sequence through the use of the ROMCC compiler. Because there is no RAM to save the stack, ROMCC used the processor's registers for this instead. By doing so, limits the code's functionality as it can keep only minimal state.

On more recent processors, Coreboot is able to use a mechanism named Cache-as-RAM (CAR) [41, 47, 53]. The terms no-fill mode [55] or no-eviction mode [76, Figure 15.1] are used by different literature interchangeably to refer to CAR. As the name suggests, CAR uses the processor's cache to save its state, as would normally be done in RAM. Although CAR is used in most recent BIOS implementations and all of the building blocks to enable it are public, to the best of our knowledge, Intel does not support nor provide any public documentation on the method as a whole. Furthermore, as with the reverse engineered memory initialization code, the code is not obvious and has been named "an esoteric area of the code" or "black magic" by Coreboot's developers.

To be able to verify and explain CAR's implementation, we have cross-referenced the available documentation with Coreboot's CAR-code for our architecture[4]. CAR works by altering the cache's operating modes [31, Volume 3A, Table 11 – 5, Cache Operating Modes]. MTRRs can to be set to change the cache its behaviour for certain memory ranges (as discussed in Section 2.2.1). CAR is in fact used by Coreboot for two purposes: (1) to save the stack; and (2) to speed up execution of the code that needs to be fetched from the BIOS chip, by loading it into the processor's cache. The first goal requires the cached data to be both readable and writeable, the second suffices with a read only region.

To enable CAR the following procedure is followed: all MTRRs are cleared, default caching type is set to uncacheable, a MTRR is set to control the memory range required, the MTRR is enabled, caching is enabled through the CPU's control register, either the region is cleared or the data is read from the BIOS chip (and thereby cached), and caching is disabled through the CPU's control register (actually enabling CAR).

4 `src/cpu/intel/model_206ax/cache_as_ram.inc` https://github.com/coreboot/coreboot/blob/e74ad21a91e33f275a7bda999b058a8390c44ae6/src/cpu/intel/model_206ax/cache_as_ram.inc

6 IMPLEMENTATION DETAILS

6.1 EXFILTRATION ROUTES

The acquisition method has been designed to acquire the memory very early in the boot sequence, because of this many, components have not yet been initialized. This poses an interesting challenge in either initializing an exfiltration route to another computer e.g. networking, or initializing a large portion of non-volatile memory e.g. a hard disk; both such that a copy of the volatile memory can be written to it and thereby secured.

As discussed in section 3.6, Schramp used post codes to extract the memory acquired. Actually working around this problem, as POST codes are typically routed by default to the Peripheral Component Interconnect (PCI) bus. However, this transfer channel is tediously slow[1], and has never been designed with large data transfers in mind. To initialize another exfiltration route, the hardware device in question needs to be enabled and a driver is required to communicate with it. Here DMA becomes of great concern, as most modern transfer methods use DMA to increase their transfer rate. DMA uses RAM to buffer the data read and written without intervention of the processor. Typically the buffer is of a specific size. Output is written directly to memory and input may be directly read from memory. However, overwriting memory is not desirable in our case and in direct conflict with the forensic soundness of the acquisition method. Still, being able to read from the peripheral can be very convenient e.g. to read the file system's structure. Therefore, we have chosen not to use DMA although the problem could perhaps be alleviated by (temporarily) saving the memory to be overwritten in e.g. the cache.

The two exfiltration routes that have been explored by us are: (1) networking using an older PCI card that contains internal buffers[2]; (2) hard disk communication. The first had already been implemented in Coreboot, yet required more platform-specific code than anticipated. Due to the fact that PCI bridges not yet having been initialized at this point in the boot sequence. However, they are required to be correctly initialized, as otherwise data will not be routed to the correct PCI device. It proves non-trivial to initialize the bridges without access to sufficient memory.

Therefore, second, the option of using the SATA controller in Programmed I/O (PIO) mode has been investigated. Invoking the

1 The same is true for the serial debug channel Coreboot provides.

2 https://blogs.coreboot.org/blog/2010/05/31/coreboot-console-over-ethernet/

SATA initialization code of Coreboot at an earlier point in time, and combining it with the SATA PIO driver of SeaBIOS enabled us to use the hard disk to save the memory acquired. The main hurdle to overcome was finding the correct I/O base-port of the SATA controller. A rate of 100 MiB per minute has been achieved and was regarded to be sufficient.

6.2 ACCESSING ALL MEMORY

The main part of Coreboot natively compiles as 32-bit code and runs in protected mode, this only allows one to address the lower 4 GiB of data (as discussed in Section 2.2). To address more memory, paging has to be enabled. Although, paging allows one to access all memory regions, the accessible memory is still limited to 4 GiB at a time. To access all memory, one additionally needs to either enable windowing or long mode. The application of windowing allows a range of low logical memory addresses (< 4 GiB) –named a window– to be translated through the paging mechanism to high physical addresses (> 4 GiB) such that the higher physical addresses may be accessed. Because this windowing introduces additional complexity in the acquisition routine, the option of running Coreboot as 64-bit code (in long mode) has been explored. However, due to Coreboot's non-trivial build system and numerous 32-bit truncated pointers, this would require significant effort to change. Although, there has been some previous work on enabling Coreboot to run as 64-bit code, this has been very experimental[3]. Therefore, our system uses windowing to acquire memory saved in high physical host addresses.

Enabling paging has also proven to be more of a challenge than anticipated; the main difficulties to overcome were the: (1) limited debugging options; (2) large number of different paging options; (3) space and alignment requirements of the paging tables; (4) combination of paging and CAR introducing unpredictable behavior. Although all of these problems can be overcome, their combination acts as a multiplier on the effort required to do so. Even more so, any of the latter three difficulties led to an unbootable system and therefore a tedious development cycle. Eventually the following options allowed us to access all memory. First, Coreboot allows for the use of a POST-card and serial debug output, the latter of which is not available until after CAR and the former only provides very rudimentary output. Second, by using Physical Address Extension (PAE) we were able to page 4 GiB of memory. Third, using PAE keeping the required paging tables small enough to save them in cache. The problem of size is increased as even small paging tables are required to be aligned on (large) page-boundaries, for this we used GCC's `aligned` variable attribute. Fourth, as all of the code is running from CAR, it is

3 https://www.coreboot.org/pipermail/coreboot/2015-January/079135.html

essential that the activation of the paging-mechanism does not interfere with the system's caching behavior. However, every page can is accompanied by several flags that define its, amongst other things, caching properties. The combination of setting the: Present, Read-/write, Accessed, Page size, and Global flags enabled us to use paging in CAR. This combination was found in the experimental 64-bit conversion previously discussed, see [31, Table 4-4. Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page] for the meaning of these bits. Furthermore, the point in the boot sequence at which paging is enabled is of great importance. Enabling it before the Memory Controller Hub (MCH) has finished initialization causes the paging mechanism to stay disabled although the processor's control registers acknowledge that it has been enabled.

6.3 MEMORY INTEGRITY CONCERNS

As described in Section 2.2 memory addresses accesses may not always be routed to data in RAM, the MCH may decide to route to addresses to different components depending on its configuration. Furthermore, some components do not come with their own memory and use (overwrite) part of the computer's RAM to function (the 'stolen' memory regions). These are only two examples of concepts that may impact the integrity of memory to be acquired.

We have identified the following components and concepts that may reduce the integrity of the memory to be acquired. As the acquisition system of a second order cold boot method may route and translate parts of memory differently than the target system, this is of particular importance in our case. The list is by no means restrictive and may be different for every platform, yet it should provide a decent overview of the concepts that may need to be taken into account:

- Memory overwrites i.e. the BIOS writing data to RAM
- Destructive POST routines i.e. destructive memory training/testing
- Reserved BIOS regions[4]
- Memory Type Range Registers (MTRRs)
- Memory regions stolen by the:
  - Internal graphics device
  - SMM
  - Intel ME (see Appendix D)
- Memory scrambling (as discussed in Part III)

4 The stock BIOS on our system reports 150 MiB to be unusable or reserved. This was observed by executing the `dmesg` command on a default Ubuntu installation.. These memory regions are, for example, not acquired by the "memimage" software released by Halderman et al. [20].

Most of these problems are inherently countered by our acquisition method. Because it acquires memory directly after memory initialization, before any other functionality is used. Only the devices stealing memory regions required explicit disabling. We chosen not to disable any of the memory training/testing routines, as this is required for the correct initialization and communication with the memory modules.

6.4 REPRODUCIBILITY

All source code to reproduce any of our results can be found as a Coreboot patch at https://review.coreboot.org/#/c/18539/.

7 VALIDATION

To validate whether the method developed is actually forensically sound, we verify whether it meets the requirements as set out in Chapter 3. Therefore, we verify whether it complies with the properties of correctness, atomicity, and integrity as defined by Vömel (and Freiling). Due to the use of the cold boot method, atomicity is implied, hence our focus is on correctness and integrity. Correctness is implied when a complete copy of all data present in the physical DIMM is acquired i.e. no memory ranges are skipped/missed. Integrity is implied when the contents of memory is not altered after the start of acquisition method e.g. due to loading the acquisition program into memory or decay due to the cold boot method. It may be the case that a property is not fulfilled completely, it is then still desirable to determine the degree of fulfillment. Performing the following experiment has allowed us to determine this.

7.1 EXPERIMENT METHODOLOGY

We employ a similar experiment as defined in [16, section II. Setup - C. Experiment]. The experiment comprises three steps: 1) filling the memory with reproducible pseudo-random data, 2) executing the acquisition method, and 3) measuring the difference between the data acquired and the expected pseudo-random data.

As opposed to the experiment defined in [16, section II. Setup - C. Experiment], we chose to differ two of the conditions. First, a different cooling method has been chosen to determine whether the original cooling technique could be improved upon (professionalized). Second, instead of executing the experiment on only relative small amounts of data e.g. 2 MiB in [16], the experiment has been performed over all data that the DIMM could hold e.g. 1 or 4 GiB such that the degree of correctness of the method as a whole could be determined. The details of the setup used have been attached as Appendix A.

7.2 COOLING TECHNIQUES

In general the cooling of memory modules is done through the use cooling spray, either dedicated freezing spray or ad-hoc upside down canisters of compressed air. By spraying this directly on the memory modules, the DIMMs are brought down to a temperature of between the $-30°$ and $-50°$ Celsius. Although this method is widely accepted

[2, 16, 20, 45], it may have severe consequences. First, the flash freezing of electrical components may cause them to malfunction. The locating of thermal faults in electrical equipment is one of the typical use-cases of freeze spray. Second, the formation of frost and subsequent condensation may cause component malfunction. Although the widespread use of freeze spray to perform the cold boot method may suggest otherwise, this is a well known side-effect of extreme cooling in the field of overclocking. Either of these consequences may result in loss of the data to be acquired.

We have, therefore, chosen to explore a different method of cooling. As noted, much experience with extreme cooling in this context may be drawn from the overclocking scene. The use of liquid nitrogen evaporation coolers is one example hereof. This type of cooler consists of a heatsink which clamps around the DIMM with a tray on top that holds the liquid nitrogen. An image of its usage is attached in Appendix A as Figure 11. As the liquid nitrogen evaporates, it cools the tray and heatsink drastically, which in turn cool the DIMM. The benefits of the method are threefold: (1) the memory can be cooled to extreme temperatures in a reduced pace as the mass of the heatsink prevents fast temperature fluctuations; (2) frost collects nearly exclusively on the heatsink instead of the DIMM. The heatsink can be removed as soon as the memory has been transplanted, reducing the risks related to condensation; and (3) The DIMM can be cooled to even extremer temperatures (around $-150°$ Celsius).

We have made no further attempt to professionalize the cooling method, but directions to explore may be: a closed circuit cooling system connected to the heat sink; or working in a humidity (and temperature) controlled environment, with enough space to execute the entire method in. The former of which is already commercially available to the overclocking community[1].

7.3 DISCUSSION OF EMPIRICAL RESULTS

The experiments we have performed, show high rates of decay for most of the DIMMs tested (irrespective of the cooling method applied). In Appendix A a table is attached containing the details of the DIMMs tested. Of the four different DIMMs tested, only DIMM `D` has led to sufficiently low and reproducible decay rates[2] to base any findings on. This has led to the fact that we are unable to draw any strong conclusions on the general susceptibility of DIMMs to data remanence; allowing more work to be done in this area. Three observations are, however, deemed worthy of a discussion.

1 See, for example, the OCC-X and Purge Case products of L&L Cooling Technologies `http://www.lnlcooling.com`

2 The overall percentage of bits that either decayed was around 0.05%

First, the sole DIMM exhibiting low decay rates is of the Small Outline DIMM (SODIMM) type, typically used in laptops. Leading us to believe that their physical construction may be more susceptible to data remanence due to their reduced power usage i.e. longer retention rates mean less refreshes in turn using less power. This suspicion is strengthened by the fact that many of the studies that are successful in reproducing the cold boot attack focus on SODIMMs [20, 45, 46][3].

Second, it is unclear when a DIMM's capacitors are being refreshed. The DIMM is controlled by the DIMM's internal finite state machine. During normal operation the MCH ensures the DIMM is refreshed. However, it is unclear how removing an reasserting power influences the state machine of a DIMM.

Third, cryogenically freezing the DIMM may cause memory initialization to fail. Cooling the DIMM to extreme temperatures has resulted in the fact the DIMM could not be initialized correctly and only being able to do so after some period of time that allowed the DIMM to warm up again.

Taken together, although prior work showed mixed results, the extreme deviation between the retention time of different memory modules of the same DDR generation was an unexpected result. This should be taken as a note of caution for anyone considering to apply the cold boot attack in a forensic context. Nonetheless, we have still been able to validate the correctness and integrity of our method by performing experiments on the sole DIMM sufficiently susceptible to the remanence effect.

7.4 CORRECTNESS

We were able to verify the correctness of the method by executing the experiment defined in Section 7.1. This has allowed us to verify all memory physically present on the DIMM is acquired, and that no memory regions are missed.

7.5 INTEGRITY

In spite of our extensive efforts to ensure no data in the DIMMs is overwritten, transplanting memory still causes certain regions of memory to be overwritten. The regions are typically located in the low address range from address 0x0 to 0x10000. We expect this behavior to be caused by the training routines or memory initialization, as the size of the memory regions differ based on the size of the DIMM initialized. Furthermore, as discussed in Section 7.3 the integrity of the method is directly related to the decay of the memory modules.

3 Although the pictures from [45] suggest otherwise, to the best of our knowledge the DDR3 modules tested in the work of Lindenlauf et al. are of the SODIMM type.

7.6 CONCLUSION

We argue that the method presented in this chapter has 'upped the ante' regarding the level of professionalization of the applicable memory acquisition methods. The application of the concept of CAR, acquiring memory directly after its initialization, disabling stolen memory regions, and experimentation with a different method of cooling, has allowed for the development of a acquisition method with a high degree of correctness.

Nevertheless, it showed to be far more difficult than expected to verify the claim of integrity, as the underlying cold boot method has shown much greater levels of deviation in the remanence period between different DIMMs than expected. Concluding, this results in a acquisition method of which its forensic soundness is directly influenced by the underlying physical properties influencing the remanence period of a DIMM.

Part III

RESEARCH

THE INTEL MEMORY SCRAMBLER

8

BACKGROUND & GOALS

8.1 BACKGROUND

As noted in Section 3.6 Cold boot, recent computer architectures include a component which scrambles the data sent over the memory bus. The reason for including this scrambler is to prevent signal integrity issues due to power spikes on the power supply. The power spikes are introduced when writing many successive ones or zeroes are written to memory [28, 2.1.6 Data Scrambling]. The switching required between a zero and a one on the memory bus would, most likely, be the cause of this; a well known relation in the field of side-channel power analysis attacks. For example, the physical representation of a logical one may be a high voltage, whereas a logical zero may be represented by a low voltage. Due to comparable deteriorating effects, similar scrambling techniques are also applied in many other high capacity wired and wireless transmission protocols e.g. SATA, USB3, PCIe and CDMA. The scrambler works by mixing-in a source of pseudo-randomness, before the data is transmitted. At the receiver's side this pseudo-randomness is removed again.

Typically the pseudo-randomness required by such a scrambler is based on the output of an LFSR (see Figure 4), due to its small implementation size and good statistical properties. The output of an LFSR is called a Pseudo-Random Binary Sequence (PRBS). An LFSR consists of a register (its state) with multiple entries, every entry contains a single bit. With every shift (or clock) of the register one bit is shifted out (output bit) and one bit is shifted back into the register (feedback bit). The feedback bit is produced through a combination of specific entries of the register. LFSRs have two main advantages; 1) LFSRs are very efficient to implement in hardware and 2) can be defined in a mathematical sense by expressing them using GF(2) (matrix) arithmetic. These two advantages have led to their wide implementation and study. The output of a LFSR repeats after a certain number of clocks, this number depends on the size of the register and feedback polynomial in use. A scrambler can be seen as a stream 'cipher' i.e. the PRBS is generated completely defined by the LFSR's internal state. The LFSR's initial state is called a seed. The idea is to XOR the data with a PRBS which consists of approximately an equal amount of ones and zeroes. This increases the number of switched bits on average, yet prevents the worst case scenario of only zeroes or ones to occur. Instead of the term cipher, henceforth, the term scram-

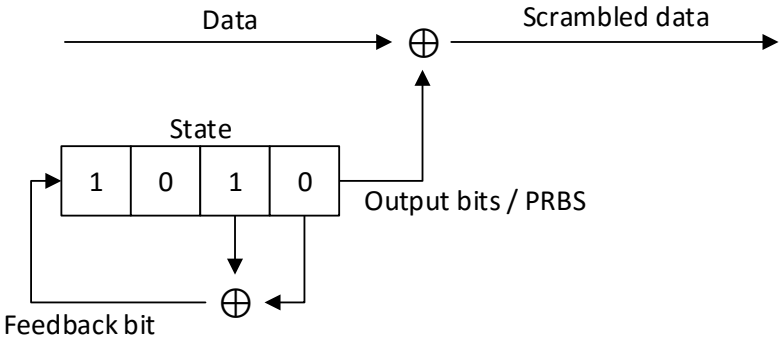


Figure 4: *Schematic overview of a typical scrambler. A PRBS is produced by an LFSR and combined with the data to obtain the scrambled data. The same system can used for descrambling. The internal state of this LFSR is* 1010 *at the moment.*

bler is used, as such a scrambling system is in no way meant to be cryptographically secure.

The main problem introduced by the scrambler is that, when the PRBS is not known, scrambled data may not easily be descrambled. Normally the scrambler performs its scrambling entirely transparent, all data sent will be received by the same scrambler with the same state. However, because the PRBS is generated based on an initial (random) seed on every boot, this may cause problems when executing the cold boot method to acquire memory. The memory to be acquired may be scrambled with the PRBS generated based on one seed, and may be descrambled with a PRBS generated based on another seed. Leaving the resulting memory image incomprehensible. This problem may intensify when different versions of scramblers are used to scramble and descramble the data.

8.2 GOALS

Two goals have been set out in determining the effects of the memory scrambler on the data. First, to reverse engineer the scrambler's workings in general such that the ideas behind its implementation are exposed. Second, to reproduce the scrambler's working, preferably using the smallest amount of data as possible. The preference of minimal data required is related to that of Kolmogorov complexity and linear complexity [58]. The concept is fairly simple, for example, the PRBS "abababababab" can be reproduced from a combination of a substring of two characters, whereas the PRBS "d24278d4a99846" may contain no such internal structure. If so, the Kolmogorov complexity of the first PRBS lower than that of the second. Inflating the idea to larger PRBSes would allow the first to be reproduced with a

smaller storage requirement than the second e.g. the first can be reconstructed with a small program whereas the second would need to be stored as a whole.

We chose to focus on a single instance of the scrambler found in the same platform, of the Ivy Bridge architecture, as used earlier in this work (see Appendix A). These two goals have allowed us to reduce the data required to reproduce the scrambler's working to only 1026 bits and have provided us with detailed insight into the implementation and the concepts behind this generation of scrambler. Our hope is that these insights may prove useful during a forensic investigation which may include different generations of scramblers. The data and code required to reproduce the working of the scrambler may be found at `https://github.com/NicoHeijningen/IntelMemoryScrambler`.

9 PRIOR & RELATED WORK

9.1 PRIOR WORK

Very little public documentation on the internal working of the Intel memory scrambler exists. Therefore, we provide an overview of relevant documentation and useful sources.

The work of Bauer, Gruhn, and Freiling [2] is the main scientific discourse on this topic. Bauer et al. were the first to have analysed the working of the memory scrambler. Through the use of a blackbox method, an attack was defined that renders the cold boot attack effective yet again. Bauer et al. note that, the PRBS they were able to distill from the scrambler actually was only very short and repeated every 64 bytes. The authors have attributed this discovery to the properties of a LFSR and enabled them to develop a "stencil" attack. The stencil attack XORs 64 bytes of known data with the scrambled data thereby retrieving the PRBS. The retrieved PRBS can then be stenciled over the remaining stenciled data to obtain the unscrambled data. The idea is similar to the cryptanalysis of data erroneously encrypted multiple times by a pad that should only have been used once ('one-time pad' crypto-system).

Coincidentally, the platforms analysed in the work of Bauer et al. and in this work are both of the Ivy Bridge architecture. Therefore, we assume that the scramblers analysed are of the same generation, such that we can correlate our findings with those of Bauer et al. We build on the work of Bauer et al. by analyzing and verifying their findings, and have made avid use of the source code released in their work. We expand their work by executing the future work proposed:

> "[...] to read out the raw key stream from a cold booted DDR3 memory module. [Would] enable brute forcing of key streams by trying different seeds, but it would also be an attack that would be significantly more difficult than what we show in this work." [2].

9.2 RELATED WORK

The datasheets regarding the Ivy Bridge microarchitecture (2012) used in our research provides no hints on the implementation of the memory scrambling [28] and is limited in technical details. However, datasheets of other architectures and generations of architectures do contain some details. For example, [33, pp. 127-128] contains the quote:

Listing 1: *Excerpt of Coreboot's memory intialization code which defines the scrambling seeds used.*

```
/* FIXME: we hardcode seeds. Do we need to use some PRNG for them
    ?
   I don't think so.  */
static u32 seeds[NUM_CHANNELS][3] = {
        {0x00009a36, 0xbafcfdcf, 0x46d1ab68},
        {0x00028bfa, 0x53fe4b49, 0x19ed5483}
};
```

> "SCRMSEED: Holds 18 bit scrambler seed value used to feed into LFSR array matrix. [...] SCRMLO: Holds 31:0 bits of scrambler parrem value used to XOR with LFSR array output. [...] SCRMHI: Holds 63:32 bits of scrambler parrem value used to XOR with LFSR array output.".

Although it is unclear what is exactly meant with a "LFSR array matrix", it suggests that an LFSR is used in combination with matrix arithmetic in GF(2). This quote can be cross-referenced to the relevant Coreboot source code[1] for our architecture; shown in Listing 1. Coreboot sets three different static seeds for every memory channel, of which the first seed is significantly smaller than the other two. Therefore, we assume henceforth that: `SCRMSEED = seeds[NUM_CHANNELS][0]`, `SCRMHI = seeds[NUM_CHANNELS][1]`, `SCRMLO = seeds[NUM_CHANNELS][2]`.

Because we can alter the Coreboot source code, we can influence the seeds used to determine the functionality of the scrambler. In turn allowing us to profile the working of the memory scrambler and the PRBS it generates. A method we return to in Section 10.2.

Starting with the Intel Nehalem microarchitecture (2008), DDR3 memory is supported through an integrated memory controller. Although, we were unable to find any technical details on the data scrambling in documents related to this architecture (or our architecture for that matter), documents related to Westmere (2010 - the die shrink of Nehalem) do discuss the data scrambling feature:

> "For every write command (CAS & WE), an LFSR is seeded using the 16 bit column address. The LFSR then generates 8 pseudo random codes to XOR with data, one for each of the 8 data transfers associated with that command. The data is then transmitted to the DRAM and stored in memory scrambled. On a subsequent read command (CAS & WE#), a second LFSR recreates the codes

1 The relevant code can be found in `src/northbridge/intel/raminit.c`: `set_scrambling_seed()` at https://github.com/coreboot/coreboot/blob/e74ad21a91e33f275a7bda999b058a8390c44ae6/src/northbridge/intel/sandybridge/raminit.c#L3784

> using the address and recovers the original data. Since memory aliasing is not allowed, data scrambling is always reversible. Each code is 16 bits wide, where the LFSR has been unrolled to "shift" 16 times per cycle, reducing correlation between different lanes or cycles. To reduce hardware cost, each bit of the pseudo random code is reused 4 times to scramble the full 64 lane bus." [50].

Although, we were unable to cross-reference this directly to the implementation under analysis, it does provide some details of the general scrambling technique applied. It is noteworthy that newer architectures may use different scrambling techniques. In a publication on the Broadwell microarchitecture (2014), Intel discusses a newer version of the memory scrambler. The document can be cross-referenced with Intel patent [68], yet again not with the implementation under analysis:

> "Broadwell changed and implemented the DDR scrambler to be programmable in a way that balances signal integrity and power requirements. Real world applications often have fairly stable data and scrambling created excessive toggling. The new scrambler provided an additional 100 – 150mW power savings." [52]

Different other patents can be found that discuss the idea of memory scrambling. The most related to our microarchitecture are Intel owned patents [51] and [13][2], the latter of which is a continuation of the first. Both patents discuss a method of scrambling data before its transmitted over a bus by using parallel LFSRs. The first patent –filed in 2007 and published in 2011– discusses a simpler form of scrambling based on an initial static seed and dynamic parameters based on the memory address. Whereas, the second patent –filed in 2009 and published in 2013– describes a more complex method of generating different LFSR polynomials based on a "boot signal". The patent does however not include any details that could directly be related with the device under analysis.

Finally, a patent that describes a device called the "TeraDIMM" has been found very relevant [67]. The TeraDIMM allows the connection of other peripherals than memory modules to the memory bus e.g. a SSD. Since the patent not only describes how such a system would work from a high level architectural point of view, but also contains some technical details on the implementation of the scrambling system. The technical details define a method to determine the different components that make up the scrambling (a point we will return to in Section 10.4.1).

2 The patent applications can also be found online and may provide additional information.

10 REVERSE ENGINEERING

10.1 PRBS ACQUISITION METHODS

To analyse the memory scrambler, the data (PRBSes) it generates needs to be acquired. First, we discuss the PRBS acquisition technique as defined by Bauer et al.. Second, the acquisition method we have developed is discussed. In the remainder of this thesis we explicitly use the terminology of 'differential' and 'plain' PRBS to differentiate (where necessary) between the PRBSes acquired using the first and second PRBS acquisition method respectively.

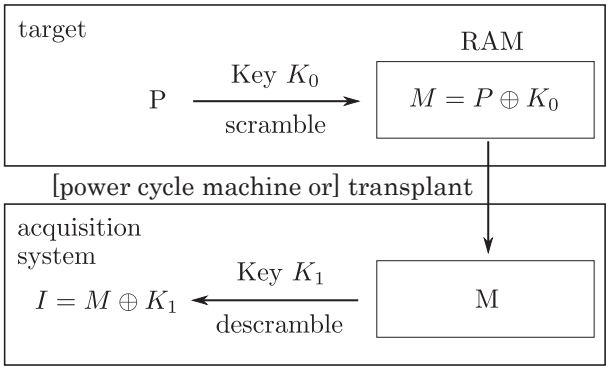
10.1.1 *Differential PRBS acquisition*

Bauer et al. [2] distill the PRBS from the scrambled data, as part of their stencil attack. Their paraphrased procedure to do so reads as follows (see also Figure 5a):

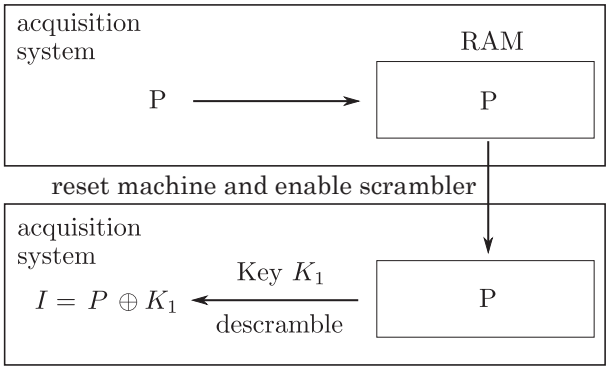
I. Turn the machine on, scrambling is enabled with random seed, the PRBS generated is K_0
II. Write known plaintext P to memory, it will be scrambled to $P \oplus K_0$
III. Turn the machine off
IV. Turn the machine on, scrambling is enabled with another random seed, the PRBS generated is K_1
V. Try to read P from memory, it will be unscrambled to $P \oplus K_0 \oplus K_1$
VI. XOR the data read with the known plaintext to retrieve the differential PRBS $K_0 \oplus K_1$ (the period of $K_0 \oplus K_1$ is 64 bytes)

10.1.2 *Plain PRBS acquisition*

Whereas Bauer et al. [2] have analysed only differential images, the acquisition method developed in Chapter 6 provides us with much more flexibility. First, instead of having the memory scrambler always enabled, we can choose to disable the memory scrambler and e.g. write unscrambled data to the DIMM. Second, the method has allowed us to change the scrambling seeds with a mere reset, instead of having to either fully power-off the machine or transplant the DIMM. The latter introduces significant decay, allowing for lossy image acquisition only. Resetting the machine eventually introduces decay, but empiric tests have shown that decay only occurs after some thousands of reset-cycles. Therefore we are able to obtain a correct copy of the



(a) Differential PRBS acquisition



(b) Plain PRBS acquisition

Figure 5: *Schematic representation of the differential acquisition method (taken from [2, Figure 3]) vs. our plain acquisition method.*

plain PRBS for many seeds. The procedure to extract the plain PRBS is as follows (see also Figure 5b):

I. Turn the machine on, with the scrambler disabled
II. Write known plaintext P to memory
III. Reset the machine, enable the scrambler with seeds of choice, the PRBS generated is K
IV. Try to read P from memory, this will be unscrambled to P $\oplus$ K
V. XOR the read data with the known plaintext to retrieve the plain PRBS K (the period of K is 1024 bytes)

10.1.3 *Differential PRBS vs. plain PRBS*

There is an interesting difference to note between the PRBSes acquired through differential and plain acquisition. We expected both PRBSes to be of the same 64 byte size and repeat itself after that. However, through plain acquisition we obtained a PRBS with a 1024 byte period. Reproducing the same differential acquisition method as Bauer et al. indeed yielded us a PRBS which repeats every 64 bytes. This observation has led to the hypothesis that the plain PRBS con-

sists of another layer of scrambling. We use the terms SCRMSEED based scrambling and address based scrambling in the remainder of this thesis to differentiate between these components. Apparently, the address based scrambling is the same for different seeds (interseed consistent), whereas the SCRMSEED based scrambling is different for every seed (intraseed consistent). Due to this the differential PRBS acquisition method of Bauer et al. only results in the SCRMSEED based scrambling component. Summarized in the following relationship, where $\oplus$ represents the bitwise XOR operation, x^y means that x is repeated y times (concatenation), $K_{SCRMSEED_0}$ and $K_{SCRMSEED_1}$ are the different 64 byte SCRMSEED based scrambling components, and $K_{address}$ is the 1024 byte address based scrambling component.

$$K_0 \oplus K_1 = (K^{16}_{SCRMSEED_0} \oplus K_{address}) \oplus (K^{16}_{SCRMSEED_1} \oplus K_{address}) = (K_{SCRMSEED_0} \oplus K_{SCRMSEED_1})^{16}$$

As our plain PRBS acquisition method is able to obtain K_0 separately from K_1, the address based scrambling does not cancel out.

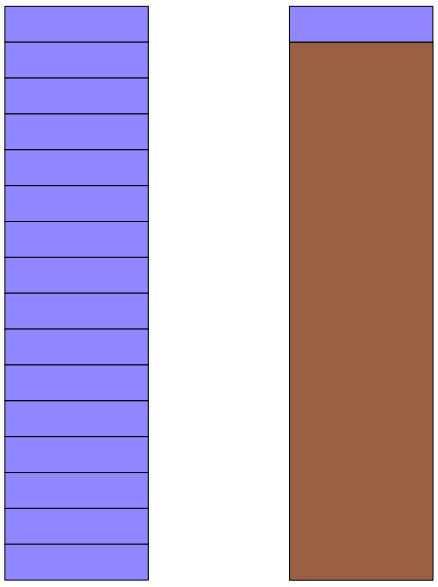
However, as our earlier engineering efforts allow us to influence the initial seeds of the scrambler, this has enabled us to execute a stronger variant of the differential PRBS acquisition method from Section 10.1.1. The procedure can be executed with seeds of choice. Now, when the differential image with SCRMSEED=0 to scramble and SCRMSEED=x ($x \in \mathbb{N}, 0 \leqslant x < 2^{18}$) to descramble is produced, the resulting (strong) differential PRBS is 64 bytes long, periodic, and identical to the first 64 bytes of the plain PRBS acquired with SCRMSEED=x (see Figure 6). We name the 64 bytes, that are the same in both the strong differential and plain PRBS, the SCRMSEED keyblock and analyse this observation in the remainder of this chapter.

10.2 PRBS INTERNAL STRUCTURE DEFINITIONS

To be able to dissect the internal structure present in the PRBS intelligibly, we introduce some terminology. An example of the definitions presented in this section, may be observed in Figure 7 and the plain PRBS in which this data occurs has been attached in its entirety as Appendix E. The PRBS may be observed from different resolutions (smaller parts), we employ the (vector) notation where y[x] means a vector named y of length x bits:

- (PRBS) word[16]; the two byte little endian[1] interpretation of the PRBS.
- (LFSR) stretch[64]; four consecutive PRBS words. A LFSR stretch contains internal structure as shown in Figure 7 i.e. $stretch_i = key_i[16,..,0] \parallel key_i[17,..,1] \parallel key_i[18,..,2] \parallel$

1 As noted in Appendix E, all pairs of bytes in the PRBSes pesented in this work have been swapped i.e. already been interpreted as 16-bit little endian integers.



(a) Strong differential PRBS

(b) Plain PRBS

Figure 6: *Schematic representation the PRBS produced by strong differential and plain PRBS acquisition using the same SCRMSEED. The SCRMSEED keyblock is represented by the blue blocks and the address based scrambling is denoted by the brown block. Each of the two PRBSes represents* 1024 *bytes of data, addressing starts at the top.*

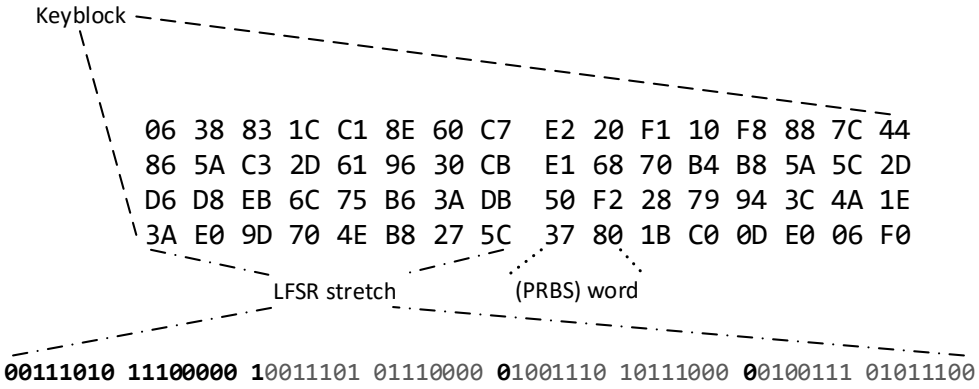


Figure 7: *Overview of the components that make up a keyblock. In this case, the second keyblock of the PRBS attached as Appendix E. The keyblock itself is printed in hexadecimal notation, one of the LFSR stretches is expanded as binary to show its LFSR like (shifting) behavior. The LFSR key is printed bold, the redundant information gray.*

$key_i[19,..,3]$. Where the $||$ operator means concatenation and i is the index of a LFSR stretch.

- (LFSR) key[19]; the 19 bits of information that make up a LFSR stretch.
- Keyblock[512]; eight consecutive LFSR stretches. All keyblocks are 512 bits aligned.

10.3 SCRMLO AND SCRMHI

As noted in Chapter 9, there are –at least– three different seeds in use that determine the functionality of the memory scrambler: SCRMSEED, SCRMHI, SCRMLO. Actually only SCRMSEED is of real interest. That is because SCRMHI and SCRMLO are together 64 bits and stenciled over or bitwise XORed with each LFSR stretch. This behavior can be cross-referenced with the Intel datasheets (discussed in Section 9.2) which ascribes the two seeds similar functionality. The exact manner in which the two seeds are applied can be deduced from Appendix F. Therefore, in the remainder of this work, we fix SCRMHI and SCRMLO to be zero and focus on SCRMSEED.

10.4 REPRODUCING THE PRBS

The most straightforward method of reproducing the workings of the memory scrambler, would be to save all the possible PRBSes and index them by the 18-bit SCRMSEED. This method is ignorant of any structure present in the PRBS and would require $2^{18} \cdot 1024 = 256$ MiB of storage space. Although, 256 MiB seems quite manageable, keep

in mind that the SCRMSEED and hence the PRBS used should[2] be different on every reboot and that a random part of the PRBS could be used for every RAM access. Therefore the only place the PRBSes can be saved, during regular operation, is in persistent memory on the processor die. In what follows, the structure present in the scrambling data is discussed in more detail, which allows us to reduce the storage space required to a more reasonable size to be stored on the processor die. Although, redundancy present between LFSR stretches and LFSR keys allows for an obvious optimization of the space requirements, we postpone this discussion until the end of this chapter.

10.4.1 *Address based scrambling (interseed consistent)*

Recalling the observation of the distinction between SCRMSEED and address based scrambling from Section 10.1.3, and cross cross-referencing this with the TerraDIMM patent [67, Data Scrambling] led us to define the following relationship. The keyblocks in the plain PRBS are a linear combination of only four special keyblocks named the generator vectors (abbreviated to gvs), these are dynamically combined depending on the memory address the keyblock starts at. Assume a linear physical memory address space, without any virtual addressing, the (physical) host address may consist of 39 bits $a = (a_{38}, ..., a_0)$ (in our case [28, Section 2.3]). Now the keyblock starting at physical host address a is defined by the following function in GF(2):

$$\texttt{address\_based\_scrambling(a)} = (a_9, a_8, a_7, a_6) \cdot \begin{pmatrix} gv_3 \\ gv_2 \\ gv_1 \\ gv_0 \end{pmatrix}$$

Thus, the input of the function is the 39-bit physical host address and the output is the 512-bit address based scrambling keyblock for that address. Only bits 9 through 6 of the input are used, and if any of the bits is a logical 1, then its respective generator vector is XORed into the keyblock; making up for the address based scrambling keyblock at that address. Because only bits 9 through 6 of the address are use, the address based scrambling keyblocks are all 512 bit aligned. This combination may be observed in Figure 8 in which the pink, yellow, red, and green blocks depict the generator vectors of (a_9, a_8, a_7, a_6) respectively. Finally, as the contribution of the SCRMSEED based scrambling with SCRMSEED=0 is all zeroes, the gen-

2 Bauer et al. discuss that some machines keep the scrambling the same over reboots, whereas others differ in scrambling. This can be cross-referenced with [32, 5.4.15 Enable Scrambling] which notes: "If scrambling is enabled by platform designer/owner, then the Firmware should load a new, random scrambler vector every cold boot."

erator vectors $(gv_3, ..., gv_0)$ may readily be found in Appendix E as the keyblocks starting at addresses 0x200, 0x100, 0x80, 0x40, respectively. As noted, the address based scrambling is applied independently from the SCRMSEED. Therefore, the four different generator vectors are the same for all possible SCRMSEEDs (interseed consistent). Hence, any data generated by the memory scrambling may be reproduced based on the four generator vectors. Together with the 2^{18} possible SCRMSEED keyblocks of the SCRMSEED based scrambling this results in a reduced storage requirement of roughly 16 MiB for reproducing all possible scrambler output.

The origin of this address based scrambling behavior lies in the mapping between physical host addresses and physical DIMM addresses[3]. The host address maps to the (DDR3) DIMM address by means of a row address, column address, and bank address (sent over the bus time-wise multiplexed one address after another). This mapping may not be obvious at first. A single bit in the host address may influence multiple bits in the DIMM address. For example different bits of the host address are XORed to obtain the physical DIMM's bank address. The authors of [56] have reverse engineered multiple of these mappings based on different architecture and DIMM combinations, and the TerraDIMM patent [67, Data Scrambling] (discussed in Chapter 8) provides an explicit example of such a mapping in relation to address based scrambling. These two works lie at the basis of ours. In our case the interesting part of the mapping is actually very straight forward. Assume that the DIMM's column address consists of 16 bits $(c_{15}, ..., c_0)$. Now we have $(a_9, ..., a_6)$=$(c_6, ..., c_3)$ i.e. four bits of the host address directly influence four bits of the column address that is sent over the DDR3 bus. The use of column addresses to base the scrambling on seems related to the Intel patents [13, 51], which note the use of the column address as a seed for the LFSR. It seems likely that an LFSR is used to generate the generator vectors, but as of yet it is unknown exactly how; a point we return to in Section 10.4.3.

10.4.2 *SCRMSEED based scrambling (intraseed consistent)*

Recall the SCRMSEED keyblocks (the blue block in Figure 8), these are generated based on the SCRMSEED and applied throughout the entire plain PRBS. The relation present within each SCRMSEED keyblock is again one of linear combinations, similar as that of address based scrambling. However, in this case the relation is based on the SCRMSEED instead of on the memory address. More specifically, the SCRMSEED keyblock of an arbitrary SCRMSEED may be reproduced by a combination of the SCRMSEED keyblocks produced by a set of special SCRMSEEDs. This set of SCRMSEEDS consists of the SCRM-

3 Recall Section 2.2.3, we refrain from using the prefix 'physical' in the remainder of this section as it is clear from this context.

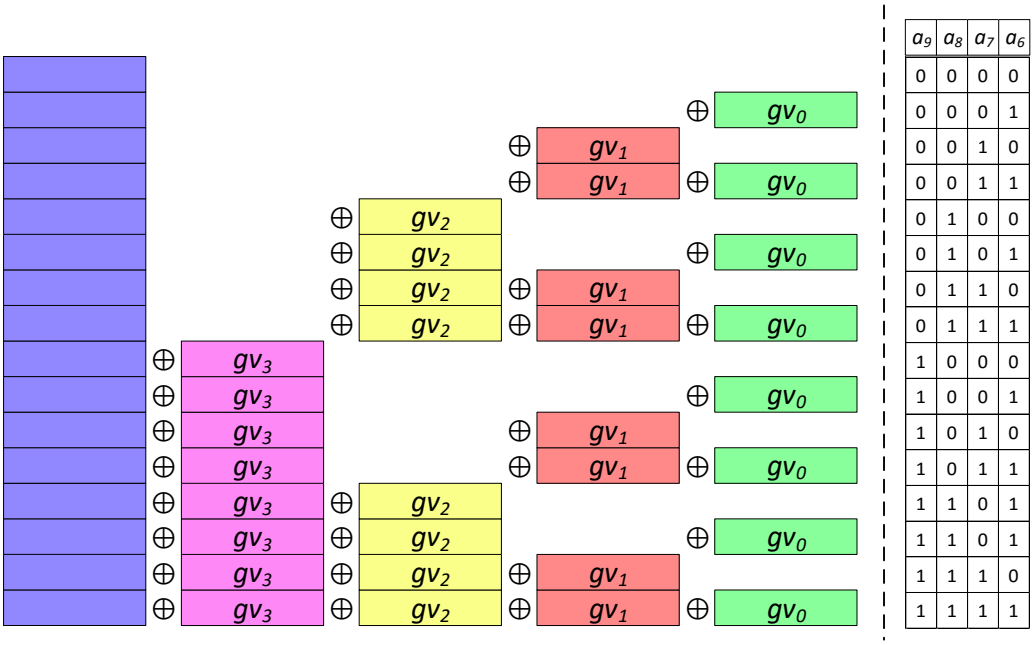


Figure 8: *Schematic representation of the PRBS produced by plain acquisition. Each vertical partition indicates a keyblock of* 64 *bytes. Whereas in Figure 6 the address based scrambling (brown) contained an unknown structure, now it is clear that it is built from a combination of different keyblocks. The addressing that influences this scrambling is provided in the table at the right hand side. The combined keyblocks represent a plain PRBS of* 1024 *bytes, addressing starts at the top.*

SEEDs with a hamming weight of one; we have named this set the toggleseeds: $(\forall x \in 2^y, y \in \mathbb{N}, 0 \leqslant y \leqslant 17)$. Hence, there are 18 different toggleseeds of which 0x0, 0x1, 0x2, 0x4, 0x8 are hexadecimal examples. We name the SCRMSEED keyblocks that the scrambler generates based on the toggleseeds the toggle vectors (abbreviated to tvs). Now the SCRMSEED keyblock of any SCRMSEED $s = (s_{17}, ..., s_0)$ is defined by the following function:

$$\texttt{SCRMSEED\_based\_scrambling}(s) = (s_{17}, ..., s_0) \cdot \begin{pmatrix} tv_{17} \\ tv_{16} \\ ... \\ tv_1 \\ tv_0 \end{pmatrix}$$

Thus, the input of the function is the 18 bit SCRMSEED and the output of the function is the 512-bit SCMRSEED based scrambling keyblock for that SCRMSEED. If any of the input bits is a logical 1, then its respective toggle vector is XORed into the equation, making up for the SCMRSEED keyblock of that SCRMSEED. Now, instead of requiring 2^{18} different SCRMSEED keyblock to reproduce the SCMRSEED based scrambling, we only require 18 of them (the toggle vectors). Together with the four generator vectors of the address based scrambling, this reduces the total space requirements for reproducing the scrambler from roughly 16 MiB to only 1408 bytes.

10.4.2.1 *Overlapping streams*

An interesting observation regarding the data of the toggle vectors, is the presence of significant redundant data. A large fraction of the data repeats sequentially, in the same order, between different toggle vectors. We have grouped the toggle vectors that overlap, highlighted the data that repeats between different toggle vectors, produced the stream of the non-redundant data (which we have named the 'overlapping stream'), and attached this in full as Appendix G. The groups of toggleseeds of which their toggle vectors contain overlapping data are: ①={0x1, 0x40, 0x400, 0x4000}, ②={0x2, 0x80, 0x800, 0x8000}, ④={0x4, 0x10, 0x100, 0x1000, 0x10000}, ⑧={0x8, 0x20, 0x200, 0x2000, 0x20000}. We leave it as an exercise for the reader to verify this in Appendix G. The 64 byte toggle vectors can be reproduced, from their overlapping stream, by starting at different offsets: {0x20, 0x8, 0x10, 0x18} and {0x20, 0x0, 0x8, 0x10, 0x18} for the overlapping streams consisting of four and five SCMRSEEDs respectively. The observation that the 18 toggle vectors can be generated from the four overlapping streams, reduces the data required to 624 bytes (recall also the generator vectors required).

Finally, it has been noted that two of the four overlapping streams can generate the other two through a transformation i.e. ① ⇒ ④ and ② ⇒ ⑧. Now, define ⓩ to be the overlapping streams of either ① or ②, denote every LFSR stretch within the overlapping stream ⓩ as stretch_0, ... stretch_{11} then the mapping from ① and ② to ④ and ⑧, respectively, is as follows:

$$\texttt{overlappingstream(ⓩ)} = \begin{pmatrix} 0\,0\,0\,0 & 1\,1\,0\,0 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,1\,1\,0 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,1\,1 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,0\,1 & 1\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,0\,0 & 1\,1\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,1\,1\,0 \\ 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,0\,1\,1 \\ 0\,0\,0\,1 & 0\,0\,0\,0 & 0\,0\,1\,1 \\ 0\,0\,0\,1 & 1\,0\,0\,0 & 0\,0\,1\,1 \\ 0\,0\,0\,1 & 1\,1\,0\,0 & 0\,0\,1\,1 \\ 0\,0\,0\,1 & 1\,1\,1\,0 & 0\,0\,1\,1 \\ 0\,0\,0\,1 & 1\,1\,1\,1 & 0\,0\,1\,1 \end{pmatrix} \cdot \begin{pmatrix} \text{stretch}_0 \\ \text{stretch}_1 \\ \text{stretch}_2 \\ \text{stretch}_3 \\ \text{stretch}_4 \\ \text{stretch}_5 \\ \text{stretch}_6 \\ \text{stretch}_7 \\ \text{stretch}_8 \\ \text{stretch}_9 \\ \text{stretch}_{10} \\ \text{stretch}_{11} \end{pmatrix}$$

Thus, the first seven LFSR stretches of the generated overlapping streams (output) are actually a XOR of two subsequent LFSR stretches of the input. For example, stretch_0 of ④ is generated by combining stretch_4 and stretch_5 of ① i.e. `20 00 10 00 08 00 04 00 = 00 01 00 00 00 00 00 00` ⊕ `20 01 10 00 08 00 04 00` (the LFSR stretches can again be found in Appendix G). After this the symmetry breaks and additional stretches are XORed into the equation. We are unable to provide an explanation for the structure of the transformation matrix, but hypothesise that this choice best suits engineering purposes.

This observation means that only two of the four overlapping streams are required. Together with the four generator vectors this reduces the space requirements from 642 to 432 bytes.

10.4.3 *PRBS LFSR stretches*

At the start of this section we noted that an LFSR stretch actually only contains 19 bits of information (the LFSR key); as defined in Section 10.2 and depicted in Figure 7. This (obvious) optimization reduces the space requirements from 432 bytes down to 1026 bits[4].

This LFSR like relation has first been observed by Bauer et al. [2, Mathematical approach]. From Figure 7 it can be observed that every

4 Two overlapping streams each consisting of eleven LFSR stretches, and four generator vectors each consisting of eight LFSR stretches; resulting in $((2 \cdot 11) + (4 \cdot 8)) \cdot 19 = 1026$ bits

keyblock consists of eight LFSR stretches each consisting of four PRBS words $w_3, ..., w_0$ in turn consisting of sixteen bits. Then as defined in [2], we have the relation:

$$((w_{(4i+j)} >> 1) \oplus p_j)\&0x7fff = w_{(4i+j+1)}$$

Where $>>$ denotes the bitwise right shift operation, & represents the bitwise AND operation, $0 \leqslant i \leqslant 7$ iterates over the LFSR stretches in a keyblock, $0 \leqslant j \leqslant 2$ iterates over the words inside an LFSR stretch, and p is a 16-bit polynomial. Bauer et al. defined this relation to be based on three polynomials. However, their work was based on solely differential PRBSes, with random values for SCRMSEED, SCRMHI, and SCRMLO on every boot. Therefore, the authors claimed that the information on the most significant bit of the adjacent word is lost (&0x7fff). Nevertheless, as we have fixed SCRMHI and SCRMLO to be zero (Section 10.3) and acquire the PRBSes through the plain acquisition method (Section 10.1.2), we should have the required information on this bit. Therefore, we may be able to determine the polynomials used in the LFSR relations. Based on the available literature [13, 33, 50, 51] we think it is reasonable to assume that this relationship is indeed introduced by a LFSR. However, we argue that the polynomials formulated by Bauer et al. are completely defined by SCRMHI and SCRMLO, as other than that the bits between the words in the relation are identical. Bauer et al. define three 16-bit polynomials, instead the relation is actually defined by two 32-bit constants (SCRMLO and SCRMHI) XORed with the whole LFSR stretch only after it has been generated (see Section 10.3).

From Figure 7 it may be observed that a LFSR stretch seems to contain four internal states of an 16-bit Fibonacci LFSR, where each state differs by one clock tick of the LFSR. Therefore, the only LFSR relationship would be the new (feedback) bit being shifted in on the second, third, and fourth word of a LFSR stretch. In comparison with other high bandwidth scrambling implementations, we believe that the direct use of the internal state of an LFSR as opposed to using the bits it outputs is atypical. This atypical usage, most likely finds its origins in optimizing the scrambler for use with the high speed DDR3 bus.

As noted, assuming a 16-bit LFSR there are three pairs of 16-bit internal states and resulting feedback bits present in every LFSR stretch. Because, the feedback bit is generated through a linear combination of the bits of the LFSR's internal state, the three pairs can be combined into a system of linear equations. Solving this system of linear equations should show the information available on the polynomial used to generate the feedback bits. In the case of the LFSR stretch shown in Figure 7 the resulting system of linear equations would be:

$$\begin{pmatrix} 0\,0\,1\,1\,1\,0\,1\,0 & 1\,1\,1\,0\,0\,0\,0\,0 \\ 1\,0\,0\,1\,1\,1\,0\,1 & 0\,1\,1\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,1\,1\,1\,0 & 1\,0\,1\,1\,1\,0\,0\,0 \end{pmatrix} \cdot (p) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Where p is an 16-bit column vector. This idea can of course be generalized to include all LFSR stretches in a keyblock. However, determining the polynomial(s) proves to be non-trivial; there are three reasons for this.

First, it is unclear what size the internal state of the LFSR is. It seemed likely that the LFSR used would also be 16-bits in length. This hypothesis was based on the work of Bauer et al., the observation that each PRBS word is 16-bits long, the use of 16-bit seeds by the Westmere documents and two Intel patents discussed in Chapter 9. However, the individual systems of equations composed of any of the overlapping streams or generator vectors, turned out to be inconsistent. Once we recalled the Intel datasheet which claims: "SCRMSEED: Holds 18-bit scrambler seed value used to feed into LFSR array matrix." [33, pp. 127-128], the assumption of a 16-bit long LFSR proved to be a naive one. Indeed, the systems of equations produced with the assumption that the LFSR's state is 18-bits are consistent. However, this increase in state size introduces a second difficulty.

Second, the number of output bits of the LFSR are limited. An LFSR with a state size of 18-bits, negatively influences the number of linear equations present in a single LFSR stretch. Again referring to the LFSR stretch of Figure 7, this has been reproduced in Figure 9. As shown there, only a single bit of information is left related to the LFSR's polynomial i.e. a combination of the LFSR's internal state and the polynomial would produce 0. Due to the lack of information, the system of equations obtained from a single overlapping stretch or a single generator vector is under determined (many different possible polynomials remaining).

Third, different polynomials are in use. As there is not enough information present in a single overlapping stretch or generator vector, it would make sense to try and add additional information to the system. A logical choice could be to combine the different generator vectors or overlapping streams and try to solve the combined system. However, this results in inconsistent systems of linear equations. Both in the case when the keyblocks of the generator vectors are combined, as when the overlapping streams are combined.

As a final note, we are familiar with the Berlekamp-Massey (BM) algorithm [48], yet this too proves troublesome to use in this setting. The algorithm requires a stream of $2n$ *consecutive* output bits from an n-bit LFSR, to be able to correctly determine the LFSR's polynomial. Assuming an 18-bit LFSR, we only obtain 8 bits of output per keyblock. Hence the BM algorithm cannot be applied.

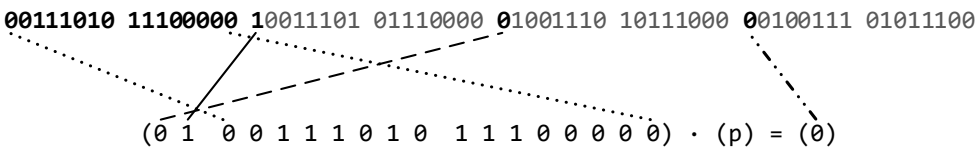


Figure 9: *How to obtain the* 18*-bit LFSR state and resulting output bit from an LFSR stretch. The LFSR stretch used is the same as in Figure 7.*

11 REMAINING PROBLEMS AND CONCLUSION

Although we have been able to determine a lot of the structure present in the output of the scrambler, it does not mean that the scrambler has been fully reverse engineered. There are some questions remaining, which allow for more work to be done on this topic:

- Where do the LFSR keys of the generator vectors and overlapping streams originate from?
- If an LFSR is used,
  - what is the size of the LFSR?
  - what are the polynomials used?
  - is there a set of constant polynomials or does the polynomial differ dynamically based on e.g. the address?
- What is the reason for the asymmetric transformation matrix used in the `overlappingstream()` function?

Concluding, we have presented the effects of the Intel memory scrambler on the data in memory in great detail. Together the relations defined in Section 10.4 lead to the fact that the PRBS for any of the SCRMSEEDs can be reproduced from only 1026 bits. Using only the four generator vectors and the two overlapping streams, all possible PRBSes can be derived. The data and code required to do so can be found at https://github.com/NicoHeijningen/IntelMemoryScrambler.

Part IV

FUTURE WORK & CONCLUSION

12 FUTURE WORK

This research could be substantiated by more results of cold boot attack in relation to different DIMMs. In such work one should make sure to include the explicit details of the memory modules tested, together with information on the experimental setup used. An explicit study into the deviation in remanence time between DIMMs and SODIMMs would be of great interest.

This research could be extended by applying it to newer architectures and different combinations of DIMMs. Examples could include newer microarchitectures such as Broadwell or Skylake or the newer DDR4 specification. The analysis of more recent architectures may prove to be even more of an hurdle due to the possible non-linearity introduced [68], and the aim of memory encryption instead of scrambling [17–19]. One could also try and solve the remaining problems related to the Ivy Bridge memory scrambler. However, we argue that this may lead to less new insights than trying to tackle a different architectures based on this work. Different DIMM combinations could introduce extra difficulties, yet these problems have been assumed to be orthogonal to the ones discussed in this work.

Furthermore, as also noted by Bauer et al., one could try to use a Field-Programmable Gate Array (FPGA) system to acquire memory. Instead of being subordinate to the Intel memory controller, one would be able to achieve the required flexibility and small TCB preferable in an forensically sound method. This would allow for the profiling of DIMMs' state machines and enable one to send DDR reset and refresh commands at will. The flexibility could allow one to use the same hardware to support the plethora of different options of the DDR standards.

13 CONCLUSION

All of the of the memory acquisition methods reviewed have their own advantages and disadvantages. However, in the context of the predefined near worst-case scenario, only the methods of BIOS modification and cold boot are deemed to be applicable. Through the combination of the two methods and by: applying the concept of CAR, acquiring memory directly after its initialization, disabling stolen memory regions, and experimentation with a different method of cooling, it showed to be possible to develop a method with a high degree of correctness. However, the results of the underlying cold boot method differ widely. Therefore, the criteria of integrity, and the general forensic soundness of the method, is directly influenced by the underlying physical properties that determine the remanence period of a DIMM.

The acquisition method, did, however, allow us to reverse engineer more of the workings of the Intel memory scrambler present in the Ivy Bridge microarchitecture. The effect of it on the data stored in memory has for a large part been exposed and only 1026 bits are required to reproduce it.

Part V

APPENDICES

A

EXPERIMENTAL SETUP

Throughout this research a system composed of the following hardware has been used:

- Gigabyte GA-B75M-D3V rev. 2.0 motherboard
- Intel Celeron G1610 processor
- A single DIMM from Table 1

The processor is of the Ivy Bridge microarchitecture (2012) and contains the memory controller [28]. The chipset is of the B75 type [27]. A picture of the setup and peripherals is attached as Figure 10, a close-up of the liquid nitrogen evaporation cooler discussed in Section 7.2 is attached as Figure 11, and the information of the DIMMs used in this research is attached as Table 1.

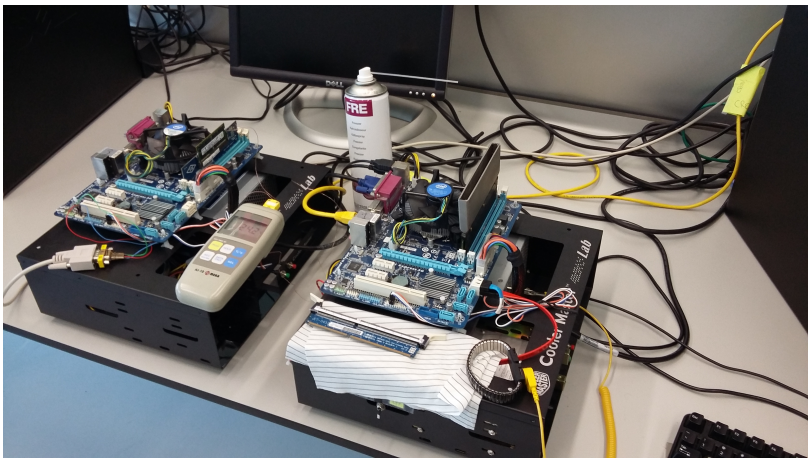


Figure 10: *Image of our setup, including: a can of freeze spray, thermometer, SODIMM adapter, DIMM interposer, liquid nitrogen evaporation cooler, and serial adapter used through the research.*

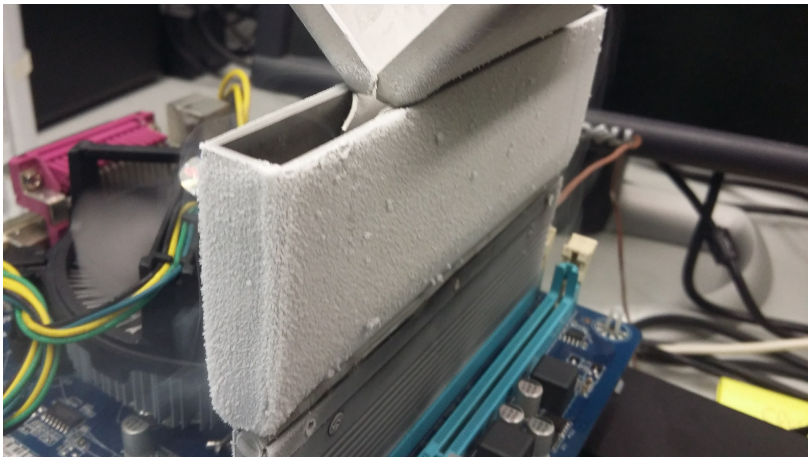


Figure 11: *Formation of frost on the liquid nitrogen evaporation cooler. Note the smaller amount of frost formation on the heatsink as opposed to the tray containing the liquid nitrogen. A funnel was used to fill the tray and a small thermocouple sandwiched between one of the DIMM's chips and heatsink to obtain relevant temperature readings.*

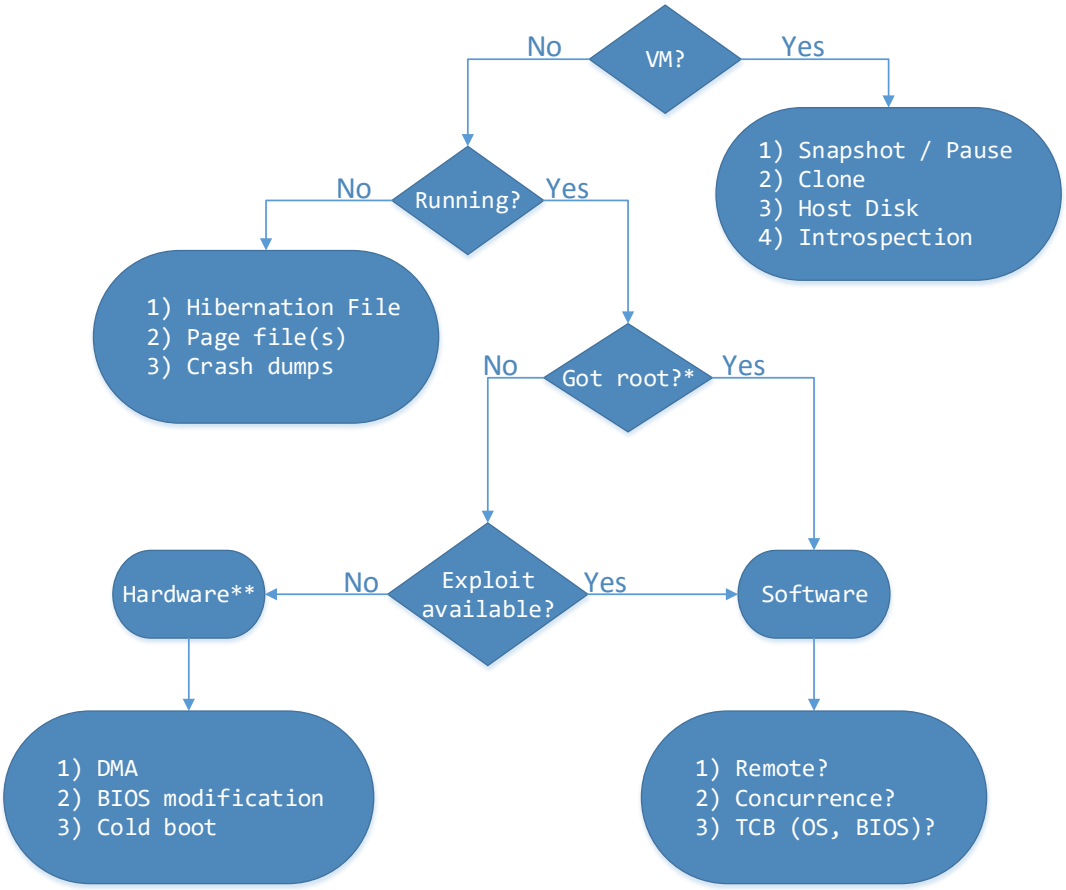
Module	Type	Manufacturer	Label	Chips (amount - type)	Capacity	Frequency (MHz)	Form factor
A	DDR3	Samsung	1GB 1Rx8 PC3 - 8500E - 07 - 10 - D0 M391B2873EH1-CF8	9 - SEC 940 HCF8 K4B1G0846E GSHE66CAC	1GB	533	DIMM
B	DDR3	Samsung	1GB 1Rx8 PC3 - 8500U - 07 - 10 - A0 M378B2873EH1-CF8	8 - SEC 934 HCF8 K4B1GO846E GS6Q85BAC	1GB	533	DIMM
C	DDR3	Kingston	99U5474 - 028, A00LF 0000006980416 - K005746 HVLWN - 69EMXL - WWWHB KVR13N9S8/4	8 - Kingston N14712 - 03 1517 S2C D5128EC4BPGGBU MHD923000D	4GB	667	DIMM
D	DDR3	Hynix	4GB 2Rx8 PC3 - 12800S - 11 - 12 - F3 HMT351S6EFR8C - PB N0 AA 1244	16 - Hynix H5TQ2G83EFR PBC 251E DT3D2583D4	4GB	800	SO-DIMM

Table 1: *List of the DIMMs used in this research as discussed in Section 7.3.*

B

FLOWCHART - MEMORY ACQUISITION OPTIONS

This flowchart shows the options that one may face during volatile memory acquisition (adapted from [44, Figure 4-1]).



*Need to be able to load kernel privileged software. Lower privilege does not allow for access to all physical memory.

**Requires physical access

C

HANDBOOK FOR THE FORENSIC INVESTIGATOR: APPLICATION OF THE COLD BOOT USING COREBOOT METHOD

The below handbook may be used to aid a forensic investigator in the correct usage of the acquisition method developed in this work.

I. Use appendix B to verify whether this is the most applicable method. **Continue only if the cold boot method is most applicable, taking into account its possible destructiveness.**
II. Obtain a reference motherboard and DIMM. Preferably both of the same type and revision.
III. Execute the method on the reference hardware first, until the method is deemed to be repeatable (consistent results).
IV. Boot the acquisition machine without a DIMM and poll for the insertion of a DIMM
V. Cool the DIMM of the target machine.
VI. Take DIMM out of the target and put it in the acquisition machine.
VII. Wait until memory is acquired and saved.
VIII. Place the DIMM back into the target machine and start a dataplacer program. Placing both randomness and the address in memory. This to be able to determine the average percentage of decay introduced, together with determining whether the DDR data lines have been swapped. Repeat until the results are consistent. If the percentage of decay is high the dataplacer can be run or the target machine and apply a first order cold boot attack. This should result in less decay and shed light on the type/generation of scrambler in place.

D

INTEL ME

We have been able to disable Intel Management Engine (ME) using two different methods: software straps and hardware straps. Straps are used to configure a component's starting stage. The software straps are used by the ME firmware. The ME firmware is loaded from the same chip as the BIOS. The layout of the data on the chip is stipulated in the so-called firmware descriptor. The descriptor does, however, also influence the working of the Management Engine. Hardware straps are used by the processor to define its initial configuration. By changing specific straps we were able to invoke a debugging mode of the ME.

The full working of Intel ME is unknown due to the lack of public documentation, but it has allowed us to boot a system without handing ME a block of RAM, which would normally not be possible.

E

NULL PRBS

The Pseudo-Random Binary Sequence (PRBS) generated when no bit in any of the seeds is set i.e. SCRMSEED = SCRMHI = SCRMLO = 0. Formatting was chosen as such to easily distinguish LFSR stretches and keyblocks. Every two LFSR stretches are prepended with the memory address. The data is printed in hexadecimal notation, where every two bytes of the PRBS have been swapped to provide for correct endianness. The generator vectors have been labeled as gv_0, gv_1, gv_0, gv_3.

SCRMSEED keyblock	0000: 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	*Intraseed consistent*
	0010: 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
	0020: 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
	0030: 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
gv_0	0040: 06 38 83 1C C1 8E 60 C7	E2 20 F1 10 F8 88 7C 44	*Interseed consistent*
	0050: 86 5A C3 2D 61 96 30 CB	E1 68 70 B4 B8 5A 5C 2D	
	0060: D6 D8 EB 6C 75 B6 3A DB	50 F2 28 79 94 3C 4A 1E	
	0070: 3A E0 9D 70 4E B8 27 5C	37 80 1B C0 0D E0 06 F0	
gv_1	0080: 52 96 29 4B 94 A5 4A 52	D2 85 69 42 B4 A1 5A 50	*Interseed consistent*
	0090: 7A B6 3D 5B 9E AD 4F 56	D1 BD E8 DE F4 6F 7A 37	
	00a0: 2E 1C 17 0E 8B 87 C5 C3	A3 7F D1 BF E8 DF 74 6F	
	00b0: 4A 12 25 09 12 84 89 42	CF 31 E7 98 F3 CC 79 E6	
	00c0: 54 AE AA 57 55 2B 2A 95	30 A5 98 52 4C 29 26 14	
	00d0: FC EC FE 76 FF 3B 7F 9D	30 D5 98 6A 4C 35 26 1A	
	00e0: F8 C4 FC 62 FE 31 FF 18	F3 8D F9 C6 7C E3 3E 71	
	00f0: 70 F2 B8 79 5C 3C AE 1E	F8 B1 FC 58 FE 2C 7F 16	
gv_2	0100: 8B 5B 45 AD A2 D6 D1 6B	0B 4D 05 A6 02 D3 81 69	*Interseed consistent*
	0110: 03 79 01 BC 80 DE C0 6F	0E 5D 87 2E C3 97 61 CB	
	0120: 6B 7B 35 BD 9A DE 4D 6F	3F D4 9F EA CF F5 67 FA	
	0130: 2F 70 17 B8 8B DC 45 EE	F3 AD 79 D6 3C EB 1E 75	
	0140: 8D 63 C6 B1 63 58 B1 AC	E9 6D F4 B6 FA 5B FD 2D	
	0150: 85 23 C2 91 E1 48 F0 A4	EF 35 F7 9A 7B CD 3D E6	
	0160: BD A3 DE D1 EF 68 77 B4	6F 26 B7 93 5B C9 2D E4	
	0170: 15 90 8A C8 C5 64 62 B2	C4 2D 62 16 31 0B 18 85	
	0180: D9 CD 6C E6 36 73 9B 39	D9 C8 6C E4 B6 72 DB 39	
	0190: 79 CF 3C E7 1E 73 8F 39	DF E0 6F F0 37 F8 1B FC	
	01a0: 45 67 22 B3 11 59 88 AC	9C AB 4E 55 27 2A 13 95	
	01b0: 65 62 32 B1 99 58 CC AC	3C 9C 9E 4E CF 27 67 93	
	01c0: DF F5 EF FA F7 FD FB FE	3B E8 9D F4 4E FA A7 7D	
	01d0: FF 95 FF CA 7F E5 BF F2	3E 88 1F 44 8F A2 47 D1	
	01e0: 93 BF C9 DF 64 EF B2 77	CC 59 66 2C B3 16 59 8B	
	01f0: 5F 82 AF C1 D7 E0 EB F0	0B 1C 85 8E C2 C7 61 63	

gv₃

```
0200: 4C 4A 26 25 93 12 49 89  F7 B1 FB D8 FD EC FE F6
0210: A0 17 50 0B A8 05 54 02  35 94 9A CA 4D 65 26 B2
0220: 59 8A AC C5 56 62 2B 31  18 B0 0C 58 86 2C C3 16
0230: AE 5F 57 2F AB 97 D5 CB  20 60 10 30 88 18 44 0C

0240: 4A 72 A5 39 52 9C 29 4E  15 91 0A C8 05 64 82 B2
0250: 26 4D 93 26 C9 93 64 C9  D4 FC EA 7E F5 3F 7A 9F
0260: 8F 52 47 A9 23 D4 11 EA  48 42 24 21 12 10 89 08
0270: 94 BF CA 5F E5 2F F2 97  17 E0 0B F0 85 F8 42 FC

0280: 1E DC 0F 6E 07 B7 03 DB  25 34 92 9A 49 4D A4 A6
0290: DA A1 6D 50 36 A8 1B 54  E4 29 72 14 B9 0A 5C 85
02a0: 77 96 BB CB DD E5 EE F2  BB CF DD E7 6E F3 B7 79
02b0: E4 4D 72 26 B9 13 5C 89  EF 51 F7 A8 7B D4 3D EA

02c0: 18 E4 8C 72 C6 39 63 1C  C7 14 63 8A B1 C5 D8 E2
02d0: 5C FB AE 7D 57 3E 2B 9F  05 41 02 A0 01 50 00 A8
02e0: A1 4E 50 A7 A8 53 D4 29  EB 3D F5 9E FA CF FD 67
02f0: DE AD EF 56 F7 AB 7B D5  D8 D1 EC 68 76 34 3B 1A

0300: C7 11 63 88 31 C4 98 E2  FC FC FE 7E FF 3F 7F 9F
0310: A3 6E 51 B7 28 DB 94 6D  3B C9 1D E4 8E F2 47 79
0320: 32 F1 99 78 CC BC 66 5E  27 64 93 B2 49 D9 A4 EC
0330: 81 2F 40 97 20 4B 90 25  D3 CD 69 E6 B4 F3 5A 79

0340: C1 29 E0 94 F0 4A F8 25  1E DC 0F 6E 07 B7 03 DB
0350: 25 34 92 9A 49 4D A4 A6  DA A1 6D 50 36 A8 1B 54
0360: E4 29 72 14 B9 0A 5C 85  77 96 BB CB DD E5 EE F2
0370: BB CF DD E7 6E F3 B7 79  E4 4D 72 26 B9 13 5C 89

0380: 95 87 4A C3 A5 61 D2 B0  2E 79 97 3C 4B 9E 25 CF
0390: D9 D8 6C EC B6 76 DB 3B  EA 74 F5 3A 7A 9D 3D 4E
03a0: 1C ED 8E 76 47 3B A3 9D  84 1B 42 0D A1 06 D0 83
03b0: CB 3D 65 9E 32 CF 19 67  1C FC 8E 7E 47 3F 23 9F

03c0: 93 BF C9 DF 64 EF B2 77  CC 59 66 2C B3 16 59 8B
03d0: 5F 82 AF C1 D7 E0 EB F0  0B 1C 85 8E C2 C7 61 63
03e0: CA 35 65 1A 32 8D 99 46  D4 E9 6A 74 35 3A 9A 9D
03f0: F1 DD F8 EE 7C 77 3E 3B  2B 7C 95 BE 4A DF 25 6F
```

Interseed consistent

F

REPRESENTATION OF SCRMLO & SCRMHI IN MEMORY

It may prove to be useful to compare this appendix with Appendix E for two reasons. First, the values of SCRMHI and SCMRLO provide an example that clarifies the change in endianness as also present in Appendix E. Second, the trivial XOR of SCRMHI and SCRMLO with every LFSR stretch can be deduced e.g. `01 23 45 67 89 AB CD EF` $\oplus$ `07 1B C6 7B 48 25 AD 28` = `06 38 83 1C C1 8E 60 C7` (second keyblock first stretch of Appendix E).

```
SCRMSEED: 0x00000000
SCRMHI: 0xcdef89ab
SCRMLO: 0x45670123

0000: 01 23 45 67 89 AB CD EF  01 23 45 67 89 AB CD EF
0010: 01 23 45 67 89 AB CD EF  01 23 45 67 89 AB CD EF
0020: 01 23 45 67 89 AB CD EF  01 23 45 67 89 AB CD EF
0030: 01 23 45 67 89 AB CD EF  01 23 45 67 89 AB CD EF

0040: 07 1B C6 7B 48 25 AD 28  E3 03 B4 77 71 23 B1 AB
0050: 87 79 86 4A E8 3D FD 24  E0 4B 35 D3 31 F1 91 C2
0060: D7 FB AE 0B FC 1D F7 34  51 D1 6D 1E 1D 97 87 F1
0070: 3B C3 D8 17 C7 13 EA B3  36 A3 5E A7 84 4B CB 1F

...
```

G

OVERLAPPING STREAMS

The eighteen toggle vectors are grouped into four overlapping stretches. The groups of toggleseeds of which their toggle vectors contain overlapping data are: ①={0x1, 0x40, 0x400, 0x4000}, ②={0x2, 0x80, 0x800, 0x8000}, ④={0x4, 0x10, 0x100, 0x1000, 0x10000}, ⑧={0x8, 0x20, 0x200, 0x2000, 0x20000}. Thus, ① and ② each consist of data from four toggle vectors and ④ and ⑧ each consist of data from five toggle vectors.

The next four pages contain the groups of toggle vectors and their overlapping stream. All toggle vectors are 64 bytes in size and printed in hexadecimal notation. The SCRMSEED used to generate the toggle vector is printed first and the toggle vector second. LFSR stretches present in more than one toggle vector have been highlighted and finally the overlapping stream of non-redundant data is shown. The order of the highlighted LFSR stretches is as follows:

```
no colour
purple
turquoise
red
yellow (consists of five concatenated LFSR stretches)
blue
green
no colour
```

SCRMSEED=0x00000001

```
00 01 00 00 00 00 00 00  20 01 10 00 08 00 04 00
20 31 90 18 48 0c 24 06  24 99 92 4c 49 26 24 93
67 d3 b3 e9 59 f4 2c fa  67 d7 b3 eb d9 f5 6c fa
e7 d1 f3 e8 79 f4 3c fa  61 cf 30 e7 18 73 8c 39
```

SCRMSEED=0x00000040

```
80 02 40 01 20 00 10 00  06 18 83 0c c1 86 e0 c3
38 00 1c 00 0e 00 07 00  00 01 00 00 00 00 00 00
20 01 10 00 08 00 04 00  20 31 90 18 48 0c 24 06
24 99 92 4c 49 26 24 93  67 d3 b3 e9 59 f4 2c fa
```

SCRMSEED=0x00000400

```
06 18 83 0c c1 86 e0 c3  38 00 1c 00 0e 00 07 00
00 01 00 00 00 00 00 00  20 01 10 00 08 00 04 00
20 31 90 18 48 0c 24 06  24 99 92 4c 49 26 24 93
67 d3 b3 e9 59 f4 2c fa  67 d7 b3 eb d9 f5 6c fa
```

SCRMSEED=0x00004000

```
38 00 1c 00 0e 00 07 00  00 01 00 00 00 00 00 00
20 01 10 00 08 00 04 00  20 31 90 18 48 0c 24 06
24 99 92 4c 49 26 24 93  67 d3 b3 e9 59 f4 2c fa
67 d7 b3 eb d9 f5 6c fa  e7 d1 f3 e8 79 f4 3c fa
```

The overlapping stream of ① is:

```
00 00 00 00 00 00 00 00  80 02 40 01 20 00 10 00
06 18 83 0c c1 86 e0 c3  38 00 1c 00 0e 00 07 00
00 01 00 00 00 00 00 00  20 01 10 00 08 00 04 00
20 31 90 18 48 0c 24 06  24 99 92 4c 49 26 24 93
67 d3 b3 e9 59 f4 2c fa  67 d7 b3 eb d9 f5 6c fa
e7 d1 f3 e8 79 f4 3c fa  61 cf 30 e7 18 73 8c 39
```

SCRMSEED=0x00000002

```
80 12 40 09 20 04 10 02  88 32 44 19 22 0c 11 06
8b 3a 45 9d 22 ce 11 67  db 38 6d 9c 36 ce 9b 67
ea b0 75 58 3a ac 9d 56  8e ba 47 5d 23 ae 91 d7
62 f2 31 79 98 bc cc 5e  8e fa c7 7d 63 be b1 df
```

SCRMSEED=0x00000080

```
88 42 44 21 a2 10 51 08  00 40 80 20 40 10 20 08
00 80 00 40 00 20 80 10  80 12 40 09 20 04 10 02
88 32 44 19 22 0c 11 06  8b 3a 45 9d 22 ce 11 67
db 38 6d 9c 36 ce 9b 67  ea b0 75 58 3a ac 9d 56
```

SCRMSEED=0x00000800

```
00 40 80 20 40 10 20 08  00 80 00 40 00 20 80 10
80 12 40 09 20 04 10 02  88 32 44 19 22 0c 11 06
8b 3a 45 9d 22 ce 11 67  db 38 6d 9c 36 ce 9b 67
ea b0 75 58 3a ac 9d 56  8e ba 47 5d 23 ae 91 d7
```

SCRMSEED=0x00008000

```
00 80 00 40 00 20 80 10  80 12 40 09 20 04 10 02
88 32 44 19 22 0c 11 06  8b 3a 45 9d 22 ce 11 67
db 38 6d 9c 36 ce 9b 67  ea b0 75 58 3a ac 9d 56
8e ba 47 5d 23 ae 91 d7  62 f2 31 79 98 bc cc 5e
```

The overlapping stream of ② is:

```
00 00 00 00 00 00 00 00  88 42 44 21 a2 10 51 08
00 40 80 20 40 10 20 08  00 80 00 40 00 20 80 10
80 12 40 09 20 04 10 02  88 32 44 19 22 0c 11 06
8b 3a 45 9d 22 ce 11 67  db 38 6d 9c 36 ce 9b 67
ea b0 75 58 3a ac 9d 56  8e ba 47 5d 23 ae 91 d7
62 f2 31 79 98 bc cc 5e  8e fa c7 7d 63 be b1 df
```

SCRMSEED=0x00000004

```
00 04 00 02 80 01 40 00  80 06 40 03 a0 01 50 00
86 1e c3 0f 61 87 b0 c3  be 1e df 0f 6f 87 b7 c3
be 1f df 0f 6f 87 b7 c3  9e 1e cf 0f 67 87 b3 c3
be 2f 5f 17 2f 8b 97 c5  9a b6 cd 5b 66 ad b3 56
```

SCRMSEED=0x00000010

```
20 00 10 00 08 00 04 00  00 30 80 18 40 0c 20 06
04 a8 02 54 01 2a 00 95  43 4a 21 a5 10 d2 08 69
00 04 00 02 80 01 40 00  80 06 40 03 a0 01 50 00
86 1e c3 0f 61 87 b0 c3  be 1e df 0f 6f 87 b7 c3
```

SCRMSEED=0x00000100

```
00 30 80 18 40 0c 20 06  04 a8 02 54 01 2a 00 95
43 4a 21 a5 10 d2 08 69  00 04 00 02 80 01 40 00
80 06 40 03 a0 01 50 00  86 1e c3 0f 61 87 b0 c3
be 1e df 0f 6f 87 b7 c3  be 1f df 0f 6f 87 b7 c3
```

SCRMSEED=0x00001000

```
04 a8 02 54 01 2a 00 95  43 4a 21 a5 10 d2 08 69
00 04 00 02 80 01 40 00  80 06 40 03 a0 01 50 00
86 1e c3 0f 61 87 b0 c3  be 1e df 0f 6f 87 b7 c3
be 1f df 0f 6f 87 b7 c3  9e 1e cf 0f 67 87 b3 c3
```

SCRMSEED=0x00010000

```
43 4a 21 a5 10 d2 08 69  00 04 00 02 80 01 40 00
80 06 40 03 a0 01 50 00  86 1e c3 0f 61 87 b0 c3
be 1e df 0f 6f 87 b7 c3  be 1f df 0f 6f 87 b7 c3
9e 1e cf 0f 67 87 b3 c3  be 2f 5f 17 2f 8b 97 c5
```

The overlapping stream of ④ is:

```
20 00 10 00 08 00 04 00  00 30 80 18 40 0c 20 06
04 a8 02 54 01 2a 00 95  43 4a 21 a5 10 d2 08 69
00 04 00 02 80 01 40 00  80 06 40 03 a0 01 50 00
86 1e c3 0f 61 87 b0 c3  be 1e df 0f 6f 87 b7 c3
be 1f df 0f 6f 87 b7 c3  9e 1e cf 0f 67 87 b3 c3
be 2f 5f 17 2f 8b 97 c5  9a b6 cd 5b 66 ad b3 56
```

```
SCRMSEED=0x00000008
64 0a 32 05 19 02 0c 81  ec 48 76 24 bb 12 5d 89
ec 08 f6 04 fb 02 7d 81  ec 88 f6 44 fb 22 fd 91
6c 9a b6 4d db 26 ed 93  e4 a8 f2 54 f9 2a fc 95
6f 92 b7 c9 db e4 ed f2  b4 aa da 55 ed 2a 76 95

SCRMSEED=0x00000020
08 20 04 10 02 08 01 04  03 08 01 84 00 c2 00 61
50 02 28 01 14 00 8a 00  31 88 18 c4 0c 62 06 31
64 0a 32 05 19 02 0c 81  ec 48 76 24 bb 12 5d 89
ec 08 f6 04 fb 02 7d 81  ec 88 f6 44 fb 22 fd 91

SCRMSEED=0x00000200
03 08 01 84 00 c2 00 61  50 02 28 01 14 00 8a 00
31 88 18 c4 0c 62 06 31  64 0a 32 05 19 02 0c 81
ec 48 76 24 bb 12 5d 89  ec 08 f6 04 fb 02 7d 81
ec 88 f6 44 fb 22 fd 91  6c 9a b6 4d db 26 ed 93

SCRMSEED=0x00002000
50 02 28 01 14 00 8a 00  31 88 18 c4 0c 62 06 31
64 0a 32 05 19 02 0c 81  ec 48 76 24 bb 12 5d 89
ec 08 f6 04 fb 02 7d 81  ec 88 f6 44 fb 22 fd 91
6c 9a b6 4d db 26 ed 93  e4 a8 f2 54 f9 2a fc 95

SCRMSEED=0x00020000
31 88 18 c4 0c 62 06 31  64 0a 32 05 19 02 0c 81
ec 48 76 24 bb 12 5d 89  ec 08 f6 04 fb 02 7d 81
ec 88 f6 44 fb 22 fd 91  6c 9a b6 4d db 26 ed 93
e4 a8 f2 54 f9 2a fc 95  6f 92 b7 c9 db e4 ed f2


The overlapping stream of ⑧ is:
08 20 04 10 02 08 01 04  03 08 01 84 00 c2 00 61
50 02 28 01 14 00 8a 00  31 88 18 c4 0c 62 06 31
64 0a 32 05 19 02 0c 81  ec 48 76 24 bb 12 5d 89
ec 08 f6 04 fb 02 7d 81  ec 88 f6 44 fb 22 fd 91
6c 9a b6 4d db 26 ed 93  e4 a8 f2 54 f9 2a fc 95
6f 92 b7 c9 db e4 ed f2  b4 aa da 55 ed 2a 76 95
```

BIBLIOGRAPHY

[1] Jacob Appelbaum. "Advanced memory forensics: The Cold Boot Attacks." Presentation at 25C3. 2008.

[2] Johannes Bauer, Michael Gruhn, and Felix C. Freiling. "Lest we forget: Cold-boot attacks on scrambled DDR3 memory." In: *Digital Investigation* 16 (2016), S65–S74.

[3] Oleksandr Bazhaniuk et al. "Attacking and Defending BIOS in 2015." Presentation at Recon. 2015.

[4] Martijn Bogaard and Ruud Schramp. "Pre-boot RAM acquisition and compression." In: (2015).

[5] Adam Boileau. "Hit by a bus: Physical access attacks with Firewire." Presentation at Ruxcon. 2006.

[6] Antonius Petrus Arnoldus Broeders. *Op zoek naar de bron: Over de grondslagen van de criminalistiek en de waardering van het forensisch bewijs*. Kluwer, 2003.

[7] Yuriy Bulygin. "Evil Maid Just Got Angrier." Presentation at CanSecWest. 2013.

[8] John Butterworth, Corey Kallenberg, and Xeno Kovah. "BIOS Chronomancy." Presentation at Black Hat USA. 2013.

[9] Richard Carbone, C. Bean, and Martin Salois. "An in-depth analysis of the cold boot attack." In: *DRDC Valcartier, Defence Research and Development, Canada, Tech. Rep* (2011).

[10] Brian D. Carrier and Joe Grand. "A hardware-based memory acquisition procedure for digital investigations." In: *Digital Investigation* 1.1 (2004), pp. 50–60.

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

[12] Maximillian Dornseif. "0wned by an iPod." Presentation at PacSec. 2004.

[13] M.C. Falconer, C.P. Mozak, and A.J. Norman. *Suppressing power supply noise using data scrambling in double data rate memory systems*. US Patent 8,503,678. 2013. URL: https://www.google.com/patents/US8503678.

[14] Ulf Frisk. "Direct Memory Attack the Kernel." Presentation at DEF CON 24. 2016.

[15] Michael Gruhn. "Forensically Sound Data Acquisition in the Age of Anti-Forensic Innocence." Dissertation. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016.

[16] Michael Gruhn and Tilo Müller. "On the practicability of cold boot attacks." In: *Availability, Reliability and Security (ARES), 2013 8th International Conference on*. IEEE. 2013, pp. 390–397.

[17] Shay Gueron. "A Memory Encryption Engine Suitable for General Purpose Processors." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 204.

[18] Shay Gueron. "Intel's SGX Memory Encryption Engine." Presentation at Real World Cryptography Conference. 2016.

[19] Shay Gueron. "Memory Encryption for General-Purpose Processors." In: *IEEE Security & Privacy* 14.6 (2016), pp. 54–62.

[20] J. Alex Halderman et al. "Lest we remember: cold-boot attacks on encryption keys." In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[21] Takeshi Hamamoto, Soichi Sugiura, and Shizuo Sawada. "On the retention time distribution of dynamic random access memory (DRAM)." In: *IEEE Transactions on Electron devices* 45.6 (1998), pp. 1300–1309.

[22] Peter Hannay and Andrew Woodward. "Cold Boot Memory Acquisition: An Investigation Into Memory Freezing and Data Retention Claims." In: *Security and Management*. 2008, pp. 620–622.

[23] John Heasman. "Implementing and Detecting an ACPI BIOS Rootkit." Presentation at Black Hat Europe. 2006.

[24] John Heasman. "Hacking the extensible firware interface." Presentation at Black Hat USA. 2007.

[25] Hoge Raad, *NJ* 1998, Vol. 404. *Schoenmakersarrest*.

[26] Trammell Hudson, Xeno Kovah, and Corey Kallenberg. "Thunderstrike 2." Presentation at Black Hat USA. 2015.

[27] Intel. *Intel® 7 Series / C216 Chipset Family Platform Controller Hub (PCH)*. 2012.

[28] Intel. *Desktop 3rd Generation Intel® Core® Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel® Celeron® Processor Family*. 2013.

[29] Intel. *New Microarchitecture for 4th Gen Intel® Core™ Processor Platforms*. 2013.

[30] Intel. *6th Generation Intel® Processor Datasheet for S-Platforms*. 2016.

[31] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2016, p. 4618.

[32] Intel. *Intel® Quark™ SoC X1000: UEFI Firmware Writer's Guide*. 2016.

[33] Intel. *N-series Intel® Pentium® Processors and Intel® Celeron® Processors*. 2016.

[34] JEDEC. *Double Data Rate (DDR) SDRAM Standard*. Version JESD79F. 2008.

[35] JEDEC. *DDR2 SDRAM Standard*. Version JESD79-2F. 2009.

[36] JEDEC. *DDR3 DIMM Label*. Version PRN09-NM4. 2009.

[37] JEDEC. *DDR3 SDRAM Standard*. Version JESD79-3F. 2012.

[38] JEDEC. *DDR4 SDRAM Standard*. Version JESD79-4A. 2013.

[39] JEDEC. *SPD Annex K - Serial Presence Detect (SPD) for DDR3 SDRAM Modules, Release 6*. Version JESD21-C. 2014.

[40] Dick James. "Recent innovations in DRAM manufacturing." In: *Advanced Semiconductor Manufacturing Conference (ASMC), 2010 IEEE/SEMI*. IEEE. 2010, pp. 264–269.

[41] H. Ke. *Method for accelerating BIOS running*. US Patent 8,037,292. 2011. URL: https://www.google.com/patents/US8037292.

[42] Kingston. *Identifying ValueRAM Memory Modules*. 2016. URL: https://www.kingston.com/en/memory/valueram/valueram_decoder.

[43] LegbaCore. "How Many Million BIOS Would You Like To Infect." Presentation at CanSecWest. 2015.

[44] Michael Hale Ligh et al. *The art of memory forensics: detecting malware and threats in windows, linux, and mac memory*. John Wiley & Sons, 2014.

[45] Simon Lindenlauf, Hans Höfken, and Marko Schuba. "Cold boot attacks on DDR2 and DDR3 SDRAM." In: *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE. 2015, pp. 287–292.

[46] Jamie Liu et al. "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms." In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 60–71.

[47] Yinghai Lu et al. *CAR: Using cache as RAM in LinuxBIOS*. 2006.

[48] James Massey. "Shift-register synthesis and BCH decoding." In: *IEEE transactions on Information Theory* 15.1 (1969), pp. 122–127.

[49] Micron. *Product Marks, Product Labels, and Packaging Labels*. 2016. URL: https://www.micron.com/resource-details/a4b348cf-e8ca-434f-906c-3d1622a62161.

[50] Praveen Mosalikanti, Chris Mozak, and Nasser Kurd. "High performance DDR architecture in Intel® Core® processors using 32nm CMOS high-K metal-gate process." In: *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*. IEEE. 2011, pp. 1–4.

[51] C.P. Mozak. *Suppressing power supply noise using data scrambling in double data rate memory systems*. US Patent 7,945,050. 2011. URL: https://www.google.com/patents/US7945050.

[52] Ankireddy Nalamalpu et al. "Broadwell: A family of IA 14nm processors." In: *2015 Symposium on VLSI Circuits (VLSI Circuits)*. IEEE. 2015, pp. C314–C315.

[53] Eswaramoorthi Nallusamy. "A Framework for Using Processor Cache as RAM (CAR)." Presentation.

[54] A. Jong-van Ormondt. *De Hoge Raad inzake forensisch bewijs. Een overzicht van de jurisprudentie*. 2004, pp. 88–98.

[55] Jürgen Pabel. "FrozenCache - Mitigating Cold-Boot-Attacks For Software-Based Full-Disk-Encryption." Presentation at 27C3. 2010.

[56] Peter Pessl et al. "DRAMA: Exploiting DRAM addressing for cross-cpu attacks." In: *Proceedings of the 25th USENIX Security Symposium*. 2016.

[57] Alessandro Reina et al. "When hardware meets software: A bulletproof solution to forensic memory acquisition." In: *Proceedings of the 28th annual computer security applications conference*. ACM. 2012, pp. 79–88.

[58] Rainer A. Rueppel. "Linear complexity and random sequences." In: *Workshop on the Theory and Application of Cryptographic Techniques*. Springer. 1985, pp. 167–188.

[59] Joanna Rutkowska. "Beyond the CPU: Defeating hardware based RAM acquisition." Presentation at Black Hat USA. 2007.

[60] Samsung. *DDR3 SDRAM Memory product guide*. 2015. URL: http://www.samsung.com/semiconductor/global/file/resourceMgmt/2016/02/DDR3_Product_guide_Dec.15_-2.pdf.

[61] Ruud Schramp. "RAM Memory Acquisition Using Live Bios Modification." Presentation at OHM2013. 2013.

[62] Russ Sevinsky. *Funderbolt: Adventures in thunderbolt DMA attacks*. Presentation at Black Hat USA. 2013.

[63] Snare. "Thunderbolt and lightning, very very frightening." Presentation at SEC-T. 2013.

[64] Peter Stuge. "coreboot: Beyond The Final Frontier." Presentation at 25C3. 2008.

[65] Jiming Sun et al. *Embedded Firmware Solutions*. Springer, 2015.

[66] Hideo Sunami. "The role of the trench capacitor in DRAM innovation." In: *IEEE Solid-State Circuits Society Newsletter* 13.1 (2008), pp. 42–44.

[67] M.L. Takefman, M. Amer, and R. Badalone. *System and method of interfacing co-processors and input/output devices via a main memory system*. US Patent 8,713,379. 2014. URL: https://www.google.com/patents/US8713379.

[68] E.L. Teoh et al. *Lower-power scrambling with improved signal integrity*. US Patent App. 14/583,623. 2016. URL: https://www.google.com/patents/US20160188523.

[69] U.S. Supreme Court, Vol. 509, No. 92-102, p. 579. *Daubert v. Merrell Dow Pharmaceuticals, Inc.* 1993.

[70] Stefan Vömel. "Forensic Acquisition and Analysis of Volatile Data in Memory." Dissertation. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[71] Stefan Vömel and Felix C Freiling. "Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition." In: *Digital Investigation* 9.2 (2012), pp. 125–137.

[72] Stefan Vömel and Johannes Stüttgen. "An evaluation platform for forensic memory acquisition software." In: *Digital Investigation* 10 (2013), S30–S40.

[73] Jiang Wang et al. "Firmware-assisted memory acquisition and analysis tools for digital forensics." In: *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*. IEEE. 2011, pp. 1–5.

[74] *Wetboek van Strafvordering. Eerste Boek. Tweede afdeling. Schadevergoeding. Titel IIIC: De deskundige. Artikel 51l.*

[75] WindowsSCOPE. *CaptureGUARD*. 2016. URL: http://www.windowsscope.com/.

[76] Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Beyond BIOS: developing with the unified extensible firmware interface*. Intel Press, 2010.