2017 IEEE International Symposium on High Performance Computer Architecture

Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors

Salessawi Ferede Yitbarek salessaf@umich.edu
Misiker Tadesse Aga misiker@umich.edu
Reetuparna Das reetudas@umich.edu
Todd Austin austin@umich.edu

University of Michigan, Ann Arbor

***Abstract*—Previous work has demonstrated that systems with unencrypted DRAM interfaces are susceptible to cold boot attacks – where the DRAM in a system is frozen to give it sufficient retention time and is then re-read after reboot, or is transferred to an attacker's machine for extracting sensitive data. This method has been shown to be an effective attack vector for extracting disk encryption keys out of locked devices. However, most modern systems incorporate some form of data scrambling into their DRAM interfaces making cold boot attacks challenging. While first added as a measure to improve signal integrity and reduce power supply noise, these scramblers today serve the added purpose of obscuring the DRAM contents. It has previously been shown that scrambled DDR3 systems do not provide meaningful protection against cold boot attacks. In this paper, we investigate the enhancements that have been introduced in DDR4 memory scramblers in the 6th generation Intel Core (Skylake) processors. We then present an attack that demonstrates these enhanced DDR4 scramblers still do not provide sufficient protection against cold boot attacks. We detail a proof-of-concept attack that extracts memory resident AES keys, including disk encryption keys.**

The limitations of memory scramblers we point out in this paper motivate the need for strong yet low-overhead full-memory encryption schemes. Existing schemes such as Intel's SGX can effectively prevent such attacks, but have overheads that may not be acceptable for performance-sensitive applications. However, it is possible to deploy a memory encryption scheme that has zero performance overhead by forgoing integrity checking and replay attack protections afforded by Intel SGX. To that end, we present analyses that confirm modern stream ciphers such as ChaCha8 are sufficiently fast that it is now possible to completely overlap keystream generation with DRAM row buffer access latency, thereby enabling the creation of strongly encrypted DRAMs with zero exposed latency. Adopting such low-overhead measures in future generation of products can effectively shut down cold boot attacks in systems where the overhead of existing memory encryption schemes is unacceptable. Furthermore, the emergence of non-volatile DIMMs that fit into DDR4 buses is going to exacerbate the risk of cold boot attacks. Hence, strong full memory encryption is going to be even more crucial on such systems.

I. Introduction

Even if DRAMs are expected to lose their content immediately after the system is powered off, studies have shown that they are capable of retaining data for several seconds after power loss – with only a fraction of data being lost. Such data retention in DRAMs has been shown to be a security risk [1]–[3], as systems that rely on disk encryption and passwords often store sensitive data in DRAM under the assumption that a reboot or removal of the DRAM will destroy the data. However, in 2008, a team of researchers demonstrated that disk encryption keys could be recovered from DDR and DDR2 DRAMs by transferring memory modules from a locked machine into an attacker's machines [3]. Since charge decay in capacitors slows down significantly at lower temperatures, they cooled the DRAMs using off-the-shelf compressed air spray cans before transferring them to another machine. This technique came to be known as a *cold boot attack.* After this demonstration, other follow-on works have explored the feasibility of cold boot attacks on a variety of DRAM-based platforms [4].

In recent years, however, it has become increasingly challenging to execute cold boot attacks or perform physical memory forensics due to the introduction of *DRAM memory scramblers*. Modern processors with DDR3 and DDR4 DRAM scramble data by XOR'ing it with a pseudo-random number before writing it to DRAM [5], [6]. These scramblers were initially introduced to mitigate the effects excessive current fluctuations on bus lines by ensuring bits on the memory bus transition nearly 50% of the time (see Section II-C). The analysis we present in this work reveals that scramblers in many modern processors (*e.g.*, Intel's Skylake) have incorporated extra features that obfuscate data. Since these features are not necessary to mitigate the electrical problems that motivated the use of scramblers in the first place, we surmise they were added as a first line of defense against cold boot attacks.

Since the details of these scramblers remain undisclosed, it has become challenging to extract and analyze DDR3 and DDR4 DRAM contents. Although multiple attempts to replicate cold boot attacks on scrambled memory failed in the past [7], [8], recent work has demonstrated a cold boot attack that bypasses DDR3 DRAM scramblers on 2nd generation Intel Core (SandyBridge) CPUs [9].

Our study reveals that DDR4 memory scramblers have been redesigned in Intel's 6th generation CPUs in a manner that provides enhanced data obfuscation over previous generation DDR3-based scramblers. While this enhanced design is resistant to attacks that have been demonstrated in the past, it is certainly not impenetrable. In this paper, we reveal details of the first DDR4-based cold boot attack that is able to successfully extract AES keys from a DDR4 DRAM connected to an Intel Skylake processor. We demonstrate this attack by extracting VeraCrypt/TrueCrypt master keys.

Our goal in this work is not to criticize the state of

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.10

memory scramblers, but to make two important observations: *i*) DRAM (including DDR4) continues to be susceptible to cold boot attacks as the scramblers do not provide sufficient confidentiality guarantees, and *ii)* modern high-throughput stream ciphers (*e.g.*, ChaCha8, CTR mode AES-128) coupled with high-speed ASIC implementations make it practical to create strongly encrypted memories that are impervious to cold boot attacks without incurring any performance penalty. In Section IV, we detail latency, area, and power trade-offs of memory encryption engine designs based on RTL simulation and synthesis. As future-generation memories will utilize dense non-volatile storage, it is becoming increasingly crucial to employ strong encryption to safe-guard the integrity of data.

In summary, we make the following contributions:

- Despite the introduction of increasingly advanced memory scramblers (*e.g., DDR3 to DDR4*), we show that these interfaces continue to be vulnerable. We demonstrate data recovery from a scrambled DDR4 DRAM, and we show how encryption keys can be stolen by descrambling memory.
- We demonstrate memory scramblers can be replaced with strong ciphers (such as ChaCha8) *without introducing any performance overheads* and with negligible power overheads.

II. Background and Motivation

A. DRAM Retention and Cold Boot Attacks

DRAMs store bits by storing charge in bit cell capacitors. Due to substrate leakage, these capacitors can lose their charge in 10s of milliseconds unless the system refreshes the bit cell. For this reason, DRAMs are conventionally expected to lose their content once a system loses power. However, studies have shown that DRAM modules can maintain a large fraction of their content after being powered down. It has been demonstrated that the bit cell capacitors can retain their charge for significantly longer periods of time (up to minutes) when the DRAM chips are super-cooled [1], [3].

This long-term retention of DRAM content poses security risks since an attacker with physical possession of a device can move the DRAM module from a secure system to an attacker-owned machine, and extract sensitive data stored in the DRAM. In 2008, Halderman *et.al.* demonstrated that DDR and DDR2 modules can retain 99.9% of the data stored in them for minutes when they are cooled down to -50^oC using an off-the-shelf can of compressed air [3]. They exploited this fact to extract sensitive data such as disk encryption keys from locked and suspended computers – an attack vector now popularly known as a “cold boot attack”.

After the demonstration of cold boot attacks, other studies have replicated the attack on additional platforms, including Android devices [4]. Another work reproduced the results from [3] and also demonstrated the feasibility of cold boot attacks on DDR3-based systems that do *not* employ any form of memory scrambling [10]. Today, many CPUs employ some form of memory scrambling that XORs data with keys generated during system boot-up. As a result, cold boot attacks have become more challenging.

B. Cold Boot Attack Mitigation Measures

To prevent extraction of encryption keys via cold boot attacks, disk encryption tools typically erase keys stored in memory immediately after a disk is unmounted. This approach can be applied on partitions other than the one the operating system is running on. While this approach reduces the attack surface, it will fail to protect disk encryption keys if a device is acquired by an attacker while disks are still mounted and the key is resident in DRAM (*e.g.*, if the machine is in sleep mode while the attacker acquires it). It should be noted that even disk encryption tools such as BitLocker that store encryption keys within trusted platform modules (TPMs) are still susceptible to cold boot attacks as the expanded keys for mounted volumes are cached in DRAM until the drive is unmounted or until the system is cleanly shutdown [11].

Solutions that store encryption keys exclusively in CPU registers have also been proposed [12], [13]. Loop-Amnesia [12] stores encryption keys in model-specific registers that are typically used by performance counters. Similarly, Tensor [13] leverages x86 debug registers for storing keys. These solutions require a patched operating system to prevent user-space access to these otherwise freely accessible registers, as they are now storing sensitive keys. Such approaches are capable of protecting disk encryption keys, but they typically suffer performance impacts since round keys must be generated before any encryption operation and subsequently erased. Previous work has shown that expanded round keys greatly simplify the task of identifying keys in memory [3], and thus, they should not reside in memory. However, due to the lack of protected on-chip storage and the limited size of registers, a large amount of sensitive data still remains in main memory, at least for a limited time, unprotected.

Full memory encryption techniques, both in hardware and software, have been suggested [14], [15]. The new Intel Software Guard Extension (SGX) includes hardware support for maintaining confidentiality and integrity of data stored in DRAM by employing strong encryption (AES) and message authentication codes (MACs). Unfortunately, SGX has been shown to incur significant performance overheads [16]. This makes such high-security solutions undesirable for latency-sensitive and bandwidth-intensive applications.

For protecting performance-sensitive applications, we need to have a solution that relaxes the security guarantees of SGX in return for better performance. We discuss these trade-offs in Section IV. The scramblers analyzed in this paper, albeit an extremely weak form of encryption, are a step in this direction. AMD has also disclosed that its upcoming CPUs will support full-memory AES encryption [17] but the performance impacts have not yet been disclosed.

Finally, newer machines with compact form factors come with their DRAM chips directly soldered on the motherboard. While this can make attacks more cumbersome, it

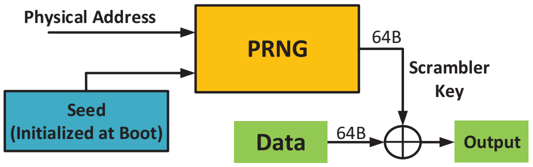


Figure 1: **High-level View of Memory Scrambling.** The scramble/descramble process is symmetric and portions of the physical address bits and a seed (generated at boot time) are used by the pseudo-random number generator (PRNG) to generate 64-byte keys.

does not fully deter them. A determined attacker can still carefully desolder the DRAM modules or boot from external media (potentially after flashing the BIOS to enable boot from external media).

C. DDR3 Memory Scramblers and Their Limitations

In older DDR and DDR2 systems, the CPU stores data in memory in plaintext form. This made capturing memory contents straightforward. With the introduction of high-speed buses however, the scrambling of DRAM data was introduced to improve signal integrity and reduce power supply noise [5].

DRAM traffic is not random and successive 1s and 0s can be observed on the data bus under normal workloads. As a result, energy can potentially be concentrated at certain frequencies or all the data lines can switch in parallel resulting in high di/dt (current fluctuations). The noise created by these phenomenon can affect signal integrity and power delivery. The Intel Core processor datasheets [6] state that by randomizing the DRAM data, potentially dangerous di/dt harmonics are eliminated. Consequently, the overall power demand of the bus becomes largely uniform.

Over time, however, these scramblers have been adapted to also provide data obfuscation, in particular with the introduction of scrambler seeds that change after each reboot. Another Intel product datasheet [18] states that its integrated memory controller has a *"DDR Data Scrambler to reduce power supply noise, improve signal integrity and to encrypt/protect the contents of memory."* These data obfuscation features thwart straightforward cold boot attacks.

Figure 1 provides a high level model of the memory scrambling unit available in current Intel CPUs. It is very similar to a symmetric encryption scheme. Before data leaves the CPU, it is XOR'd with pseudo-random numbers. We have observed these pseudo-random numbers to be a function of the *address bits* and *a pseudo-random number generated at boot time*. When data is read back from DRAM, it is XOR'd with the same pseudo-random number to recover the original data.

While Intel's datasheets do not provide any additional details about their DDR3-based data scrambler architecture, their 2011 publication [5] discloses that Linear Feedback Shift Registers (LFSRs) are used as pseudo-random number generators (PRNGs) by the scrambler implemented in the Westmere microarchitecture. An LFSR is a simple hardware component commonly used to generate pseudo-random numbers. It consists of a shift register and a feedback function that sets the leftmost bit of the shift register. The feedback function is conventionally an XOR of some of the bits of the shift register. Different random number sequences can be generated by varying the initial state of the LFSR, register width, and the bits that are XOR'd together. The Intel publication [5] also discloses that the LFSRs are seeded using a portion of the address bits. This reduces correlations between memory blocks containing the same data values.

Recent successful attempts to reverse engineer Intel's DDR3-based scrambler revealed a number of characteristics of the scrambler that led to a successful cold boot attack of a DDR3-based system [9]. It has been shown that only 16 distinct keys are generated per memory channel for scrambling data. These keys are reused numerous times to scramble the entire memory space, thus creating the possibility of correlations between memory blocks with the same data (see Figure 3b).

The most important property that enables bypassing of DDR3 scramblers stems from the fact that re-reading data from a scrambled memory after reboot (or using a second identical CPU) factors out (cancels out) portions of the keystream. As a result of this factoring, the entire memory will appear as having been scrambled using a single key (see Figure 3c). This essentially ends up resembling a block cipher operating in electronic codebook (ECB) mode. This clearly makes, de-scrambling DRAM using a second identical system significantly straightforward.

The DDR4 controllers that we studied in this work eliminate this property. However, as we will demonstrate, they continue to be susceptible to cold boot attacks. Our study reveals that while additional levels of obfuscation have been introduced for DDR4 interfaces, the protections are still weak enough to permit recovery of sensitive data via cold boot attacks.

III. Cold Boot Attacks on DDR4

In this section, we present a successful cold boot attack on an Intel Skylake-based system with DDR4 DRAM. In the first subsection, we detail our experimental framework for analyzing scramblers and implementing cold boot attacks. We then present our understanding of the DDR4 scramblers based on our analysis efforts, and finally we give details of a successful recovery of a VeraCrypt/TrueCrypt AES encrypted drive volume key through a cold boot attack.

A. Analysis Framework

Since the scramblers implemented in modern CPUs are not publicly documented, we needed to empirically analyze the data transformations applied by the memory controller before attempting to identify its limitations. For this study, we analyzed data stored by the DDR4 memory controllers integrated in Intel's 6^{th} Generation Core Processors. For

CPU Model	Microarchitecture	Launch Date
i5-2540M (DDR3)	SandyBridge	Q1, 2011
i5-2430M (DDR3)	SandyBridge	Q4, 2011
i7-3540M (DDR3)	IvyBridge	Q1, 2013
i5-6400 (DDR4)	Skylake	Q3, 2015
i5-6600K (DDR4)	Skylake	Q3, 2015

Table I: **CPU Models of Tested Machines.** In this paper, we analyzed the DDR3 and DDR4 based memory scramblers of the listed processors. We present a successful cold boot attack on the listed DDR4-based systems.

comparison purposes, we also analyzed scramblers in multiple generations of DDR3 controllers. We performed this analysis on multiple notebooks and a desktop computer. The CPUs we have analyzed are given in Table I.

All data that is eventually written to DRAM passes through the scrambler. Similarly, all data that is read by software is first passed through the descrambler and regular software cannot see the raw scrambled data. This scrambling/descrambling algorithm is implemented inside the memory controller, which cannot be directly accessed. Hence, we needed to devise a mechanism for capturing and observing the raw output of the memory scrambler. We did this using two approaches. For the DDR4 DRAMs, we relied on a motherboard that enabled us to switch the scramblers on and off through the BIOS configuration menus. However, the DDR3-based systems we used for comparative analysis do not expose a mechanism for controlling the scrambler. Hence, we relied on an external FPGA-based system to directly access memory contents. On the FPGA board we can read and write any raw (unscrambled) data. For our experiments, we used the Xilinx VC709 board with Virtex-7 FPGA to write unscrambled data to the DRAM.

To extract the scrambler keys, we implemented a "reverse cold boot attack" on a memory filled with all zeros. We use the mechanisms we just described to write *unscrambled* zeros to a DRAM module. Given that the final step of scrambling is XOR'ing the scramble key with the data, we can discover the keys by initially filling all memory with *unscrambled* zeros and then re-reading the data with the scrambler turned-on. In this case note that when the zeros are read back through the descrambler, it will attempt to descramble the data using the scrambler keys and we are actually reading the scrambler keys themselves (*i.e.*, $0 \oplus key$). Based on this approach, we extract the scrambler keys using the following steps:

1) On a system where scrambling is disabled, we fill the entire memory with raw (unscrambled) zeros.
2) We freeze the DRAM and transfer it onto the motherboard of the system we are analyzing.
3) We boot scrambled system and read the raw zero values from memory using our custom GRUB module that runs on the bare hardware.

The resulting memory image retrieved by the GRUB module is filled with scrambler keys (since a scrambler key XOR'd with zero yields the key). The program we run to extract the memory dump has no operating system or virtual memory manager running underneath it. Hence, we have full view of DRAM contents while introducing minimal pollution to the memory contents. Note that this procedure is the reverse of a cold boot attack, since in this situation we want to inject known data *into* a scrambled system.

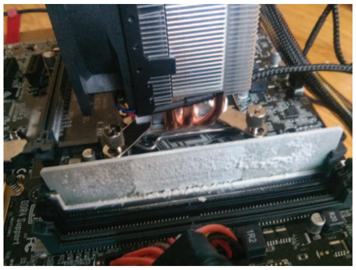


Figure 2: **Cold Boot Attack on DDR4 DRAM.** This photo shows the DRAM in one of our DDR4-based systems. The DRAM is filled with data scrambled by the memory interface of an Intel Skylake-based CPU. The memory has been cooled to $-25°C$, and it will next be moved to a separate system where its contents will be descrambled.

Instead of filling the DRAM with zeros, we can alternatively begin by allowing the DRAM to fully decay to its ground state. We can then read out the value each DRAM block assumes at this ground state *with the scrambler turned off*. Note that portions the DRAM cells decay to a zero while others decay to a one. After this initial "profiling" stage, we can boot into a scrambled system with the fully decayed DRAM and read out this known data (*i.e.,* the ground state values) through the scrambler. Unlike the technique where we fill the memory with zeros, we do *not* have to worry about bit decay that might occur in midst of the experiment.

Later in our research, we acquired a DDR4-based motherboard that allowed us to reboot an initially scrambled machine with the memory scramblers turned off – without destroying the scrambled DRAM contents from the previous boot cycle. Hence we were able to study the data transformations made by the scrambler by simply writing scrambled data to memory and reading it back out on the next boot cycle with the scrambler turned off. It should be noted that this setup was used to speed up our analysis, and the cold boot attacks detailed later in this section were indeed tested by transporting a frozen DDR4 DRAM across two machines. Figure 2 shows the frozen DDR4 DRAM on the scrambled machine's motherboard, prior to being pulled out and re-socketed into the motherboard of a machine with a disabled scrambler.

B. Analysis of a DDR4 Scrambler

Using the framework detailed in the previous section, we extracted the scrambler keys used by the CPUs we analyzed. After analyzing the extracted keys and their characteristics throughout memory and between subsequent boots of the system, we were able to make the following observations for the DDR4 memory scramblers in Intel's Skylake CPUs:

- A memory channel is scrambled using a total of 4096 distinct 64-byte keys (in contrast to just 16 keys in the DDR3-base memory systems). While visible correlations could exist for the same data in different 64-byte blocks, their probability of occurrence compared to DDR3-based DRAM is reduced by a factor of 256. This effect can be seen by comparing Figures 3b and 3d.
- These 4096 keys generated for every channel are all reset after system reboot. However, BIOS from certain vendors do not reset the scrambler seed every boot cycle and the same set of scrambler keys are reused after reboot.
- Unlike older DDR3-based scramblers, reading back data on an identical machine after reboot (*i.e.*, after the scrambler is reset) does not result in the entire memory being scrambled with a single 64-byte key. This can be seen by comparing Figures 3c) and 3e). That is, the XOR of all the corresponding current keys and the previous keys does not result in a single universal 64-byte key. As such, cold attacks devised for scrambled DDR3 DRAM are not applicable to Intel Skylake based DDR4 systems as they relied on discovering a single 64-byte universal key.
- The scrambler keys appear to be generated using a combination of a scrambler seed generated at boot time by the BIOS and portions of the physical address bits. Consequently, different memory blocks that share a scrambler key continue to share a scrambler key after reboot.

To descramble a DDR4 DRAM during a cold boot attack, we need a mechanism to recover the scrambler keys solely from data captured out of a scrambled DRAM. Since a zero value XOR'd with the scrambler key will result in the key itself, memory blocks with zeros written to them will contain the actual scrambler keys. It has been shown that zeros occur more frequently than most other individual values in memory – an occurrence which has been a basis for multiple proposed memory compression algorithms.

Therefore, the challenge lies in identifying which memory blocks contain scrambler keys (*i.e.*, are zero'd memory blocks). Previous attacks on DDR3 systems only had to extract one key for each channel and hence relied on straightforward frequency analysis [9]. However, due to the large number of keys at play in the newer systems, we cannot reliably use simple frequency analysis.

The key to identifying a scrambler key in a memory dump lies in an observation that we made regarding properties of the scrambler keys. After extracting the scrambler keys using the technique detailed above, we were able to identify *invariants on the scrambler keys* that we used to form a *scrambler key litmus test*. These litmus tests allowed us to identify zero-filled blocks in memory images that reveal a scrambler key. The invariants are between byte pairs in a 64-byte scrambler key.

These invariants are better understood by partitioning the 64-byte memory block into 2-byte words. In the expressions below, *K[i:j] represents bytes within a 64-byte scrambler key starting at byte i and ending at byte j*.

Using this notation we can describe relationships that hold true within any **64-byte scrambler key**:

$$K[i:i{+}1] \oplus K[i{+}2:i{+}3] = K[i{+}8:i{+}9] \oplus K[i{+}10:i{+}11]$$

$$K[i:i{+}1] \oplus K[i{+}4:i{+}5] = K[i{+}8:i{+}9] \oplus K[i{+}12:i{+}13]$$

$$K[i:i{+}1] \oplus K[i{+}6:i{+}7] = K[i{+}8:i{+}9] \oplus K[i{+}14:i{+}15]$$

$$K[i{+}2:i{+}3] \oplus K[i{+}4:i{+}5] = K[i{+}10:i{+}11] \oplus K[i{+}12:i{+}13]$$

$for\ i = 0, 16, 32, 48\ (i.e.,\ for\ each$ **16-byte aligned words**$)$

While it is possible to setup a system of boolean equations using the above expressions and attempt to find candidate solutions for the unscrambled text, we have found that approach to be computationally intensive. Instead, we use these expressions as a litmus test to check if a given memory block in a true DDR4 memory dump is a likely 64-byte zero-value block (thus being a scrambler key exposed in the memory dump). Even on a heavily loaded system, we were able to mine all scrambler keys by running the tests on less than 16MB of the memory dump. Consequently, a small memory dump can quickly produce all of the keys used. These litmus tests are still valid and can extract keys required for descrambling even when data is read back through a scrambler with a different set of keys. As a result, an attacker does not require a machine with a disabled scrambler.

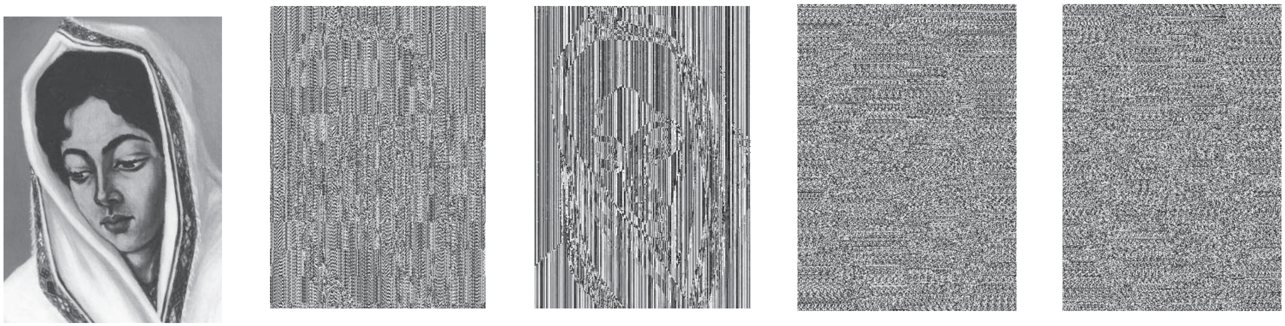
It should be noted that portions of the bits stored in the DRAM can decay while the DRAM is being transported to the attacker's machine. We will discuss how we tolerate such data loss in the next subsection.

> **Key Idea 1:** The DDR4 scrambler generates 4096 distinct scrambler keys *for each channel*. These keys can be mined from a memory dump by testing memory blocks against a set of litmus tests. These tests can be performed in a manner that is resilient to modest bit flips.

C. Disk Encryption Key Recovery from a DDR4 Memory

We now turn our attention to designing a cold boot attack on a Skylake-based DDR4 system. In this attack, the scrambled memory dump is obtained by extracting a frozen DDR4 DRAM from the secure system, and placing it in a system with a disabled scrambler where it can be dumped to disk. The proof-of-concept attack we present here focuses on recovering the AES encryption keys, specifically those used to decrypt a secure TrueCrypt/VeraCrypt disk volume on a Linux machine. However, it can be extended to extract any other information.

Attack Model: The attack we present here assumes the attacker has no knowledge of which memory blocks share the same scrambler key, and the attacker has no specific knowledge of the unscrambled contents in the scrambled memory. These assumptions helps to demonstrate that simple permutations of the random number generators and key mapping schemes (as different generations of DDR3 controllers have done in the past) would not affect this



(a) Original Image (b) Scrambled DDR3 Data (c) Scrambled DDR3 Data Read Back After Reboot (d) Scrambled DDR4 Data (e) Scrambled DDR4 Data Read Back After Reboot

Figure 3: **Visual Comparison of DDR3 and DDR4 Scramblers.** Due to the larger key pool used in the Skylake DDR4 scramblers, repeated data in memory reveal fewer correlations compared to DDR3 (compare (b) and (d)). Additionally, unlike DDR3, portions of the key are not factored out in the DDR4 scramblers when data is loaded back using a different seed (compare (c) and (e)). Overall, DDR4 memory achieves better data obfuscation.

attack's ability to recover sensitive information. If a second machine is used for dumping the memory image (instead of rebooting the same machine), then the attacker must use a CPU that is the same generation as the one being attacked. This restriction is important as different generations of Intel CPUs can have different physical address to channel, rank, bank, and row mappings. As noted above, the scrambler on the attacker's machine or on the machine being attacked does not need to be turned off when capturing memory images.

Like previous cold boot attacks on unscrambled memory systems (*e.g., DDR and DDR2*), we search for an expanded AES key, which has special properties that allow it to be easily distinguished from all other data in the system [3]. Our search, however, is complicated by the fact that an AES round keys can span four 64-byte memory blocks, thereby requiring us to guess four different scrambler keys from a total of 4096 possibilities (8192 for a dual channel system) to fully descramble the keytable. If brute forced, this would result in 2^{48} different combinations for each set of four memory blocks on a single channel system. To work around this limitation, we modified the algorithm in [3] to recover AES keys from a scrambled memory without having to descramble more than a single 64-byte block at a time. Fortunately (for the attacker, and unfortunately for all else) we can test if a given 64-byte memory block contains portions of the AES round keys. Thus, we can form an *AES key litmus test* for a 64-byte memory block that, if it holds true, tells us if we are in middle of contiguous memory blocks that contain AES round keys.

Specifically, our attack algorithm works as follows on a scrambled DDR4 memory dump:

1) Scan the memory image for 64-byte aligned, zero-filled memory blocks that reveal scrambler keys directly. These candidate keys, K, are located when they pass the *scrambler key litmus test* detailed in previous section. Note that not all of the candidate keys K are scrambler keys. However, many of them are and those that occur more frequently are likely keys.
2) Using the candidate scrambler keys, K, gathered in the previous step, descramble individual memory blocks in the dump with all keys K, looking for descrambled memory blocks that pass the 64-byte block *AES key litmus test* (explained in detail below).
3) For all descrambled memory blocks that pass the 64-byte block AES key litmus test (S_i, K_j), repeat Step 3 on neighboring blocks until a complete set of AES round keys have been located.
4) When a complete set of AES round keys has been found, recover the secret AES key from the head of the table.

AES Key Litmus Test: The standard AES algorithm can operate with a key length of 128, 192, or 256 bits. However, the key supplied to the algorithm is *expanded* to form a longer key using an algorithm that only depends on the key. This expansion is necessary since the algorithm encrypts data by applying a round function multiple times, using a different key each time. For example, in AES-256, a 256-bit key is expanded to generate 16-bit keys for each of the 14 rounds – forming a total of 240 bytes. These round keys are normally computed once and stored in memory.

The AES key search algorithm described in [3] works by sliding a search window across a stream of bytes looking for an expanded AES key. However, this algorithm assumes the full memory image is descrambled ahead of time. As a result, it picks 256-bits of data (for AES-256) and applies the standard key expansion algorithm. Similar to their algorithm, we rely on the contiguous storage of round key in memory for recovering keys. However, we do not require the memory image to be fully descrambled for the algorithm to work. Our modified algorithm is based on one straightforward insight: in a contiguous memory region containing AES round keys, at least 3 consecutive round keys will reside in a 64-byte memory block, regardless of how the key is aligned in memory. An example is shown in Figure 4. Except the first memory block, all the others contain 3 full round keys (*e.g.*, the second memory block in the figure contains complete keys for rounds 4, 5, and, 6). If the data structure storing

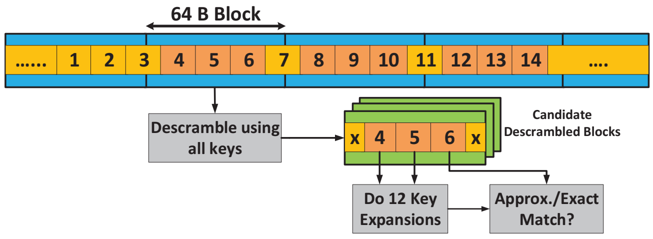


Figure 4: **Scanning Memory for AES Round Keys.** Our attack locates the AES round keys used to decrypt the disk. To locate the round keys, we descramble a 64-byte block using all (thousands) of the candidate scrambler keys. If the block contains portions of AES round keys, it will be possible to successfully run at least one iteration of the AES key expansion algorithm (since three round keys fit in 64 bytes). Since we do not know which three round keys lie in the block, we need to try all 12 possible expansions (*e.g.*, 1,2,3 and 2,3,4, *etc.*).

the key happens to be aligned to 64 byte boundaries, 4 round keys would end up in a single block. For all other cases, however, 3 of the keys would appear unfragmented in a single memory block.

Due to this guarantee, it is possible to check if a single descrambled memory block is storing portions of an expanded encryption key. We first create descrambled candidate blocks by XOR'ing a scrambled memory block with all the candidate scrambler keys. Then, for each candidate descrambled block, we take 256 bits of data (with varied offsets) and pass it through the key expansion algorithm. Since we do not know which round keys we are going to encounter, we cannot simply apply the standard key expansion algorithm. Instead, we do all 12 possible partial expansions for AES-256 by executing the key expansion algorithm starting at each of the 12 different rounds. The expansion results are then checked against the stream of bytes adjacent to the 32 bytes we just expanded. The fact that multiple contiguous blocks will pass this check when an expanded key is encountered enables us to be resilient to bit decay that might have occurred while acquiring the memory image.

Once we encounter a series of contiguous memory blocks containing AES round keys, we check blocks at the boundaries to extract any remaining bytes that are part of the key. In Figure 4 for example, bytes from keys for rounds 1 and 2 need to be extracted from the memory block that appears immediately before the group of memory blocks we have identified. This step might not be necessary depending on the alignment of the data structure. By performing this scan on the memory dump, we were able to successfully extract AES-256 keys. For AES-128 and AES-192, we can run the same algorithm using their respective key expansion algorithms.

Tolerating Data Loss: Due to the possibility of bit decay while transferring cooled DRAM, in all the algorithms described above, we measure hamming distance to test equality instead of relying on a simple bit-by-bit comparison. Additionally, since a single scrambler keystream appears multiple times inside a memory dump, we are able to filter out modest bit flips with minimal effort.

Attack Performance: Our implementation speeds up the search process by leveraging the Intel AES instruction set extensions (AES-NI). AES-NI provides us with hardware support for performing fast key expansion. Using this algorithm we were able to scan 100MBs of memory using a single core in just 2 hours. Furthermore, since the task is fully parallelizable, we can analyze gigabytes of data in a matter of hours using multiple machines. For example, using a machine with an eight-core Intel Xeon D1541 CPU, we are able to fully search an 8 GB DDR4 DRAM image in just over 21 hours.

D. Physical Characteristics of DDR4 DRAM

DRAM modules manufactured today are much denser than the DRAMs originally attacked in [3]. To assess the feasibility of cold boot attacks on today's denser and smaller components, we measured the retention time of five DDR3 and two DDR4 modules from various manufacturers. At normal operating temperatures, a significant fraction of the data is lost within 3 seconds of losing power. To measure retention characteristics at reduced temperatures, we sprayed the DRAM with an off-the-shelf compressed gas duster to super-cool them. The super-cooled the DRAMs reached a temperature of approximately $-25^{\circ}C$. In all cases, we observed that the modules are capable of retaining 90%-99% of their charges if transferred to another machine in approximately 5 seconds after being unplugged from a live system. Interestingly, one of the DDR3 modules we tested leaked data faster than the newer DDR4 modules. The algorithms we presented in this work are resilient to these modest bit flips.

It should be noted that DRAM manufacturers cannot reduce the "volume" of capacitors beyond a 10s of femto Farads without compromising reliability or significantly increasing the DRAM refresh rate (which has remained fixed over many previous generations of DRAM). For this reason, we believe that DRAM modules will continue to be susceptible to cold boot attacks for the foreseeable future. More importantly, the emergence of non-volatile DIMMs that fit into DDR4 buses is going to exacerbate the risk of cold boot attacks. Hence, strong memory encryption is going to be more crucial on these systems.

IV. Replacing Scramblers with Strong Ciphers

Our results demonstrate that current memory scramblers cannot provide meaningful protection against cold boot attacks since they use PRNGs that are not cryptographically secure. On the other hand, replacing memory scramblers with cryptographically strong cipher engines (*e.g.*, ChaCha, AES) can provide significantly better protection against cold boot attacks, since any cold boot attack would require brute-force decryption of the strong cipher. Both strong encryption and scrambling aim to transform data into highly random bit streams. Hence, cipher engines will also mitigate the electrical problems that led to the initial introduction of memory scramblers (see Section II-C). By definition, a secure encryption algorithm is indistinguishable from randomly generated data, which is the desirable characteristic of data being transmitted on a high-speed bus.

Encrypting memory contents is going to be even more important in the near future due to the imminent adoption of dense non-volatile RAM (NVRAM) DIMMs [19]. These DIMMs are being designed as a stand-alone storage or as a hardware managed backing store for DRAMs. In either case, these emerging memory technologies can hold many secrets, and the attacker would not even need to cool down the modules before transferring data to a separate machine.

A. State of Memory Encryption in Current Products

CPU vendors, most notably Intel and AMD, have started integrating memory encryption modules into their products [6], [17]. These security solutions can effectively shutdown cold boot attacks.

However, one major concern that arises with the introduction of strong memory encryption into a system is that it might incur extra latency on DRAM reads. For example, it has been shown that the strong confidentiality and integrity guarantees provided by Intel's Software Guard Extension (SGX) come with a performance penalty ranging from a few percents to 12x depending on the access pattern and working set size [8], [16][1]. This significant overhead is partly due to the fact that SGX augments strong encryption with integrity checking and code isolation, and there is no mechanism for software developers to selectively disable some of these features. Furthermore, strong memory encryption is employed only for applications that explicitly setup a secure memory region using the new SGX instruction set. The need for software modification and the associated performance overhead with solutions such as SGX can possibly limit the number of applications that leverage such strong protections.

Our aim in this section is to show that it is possible to replace memory scramblers with low-power, low-latency, and high-throughput cipher engines that introduce zero extra latency on memory reads. By forgoing integrity checking and replay attack protection afforded by Intel SGX, we show that is possible to provide protections against cold boot attacks for the entire memory with no performance overhead.

In addition to SGX there have been multiple proposals to enforce integrity, confidentiality, and oblivious execution [20]–[22]. The optimization and overhead exploration we discussed in this paper would complement such efforts that target stronger attack models beyond cold boot attacks.

B. Low Overhead Memory Encryption

In this section, we argue that power-efficient cipher engines can be used to transparently replace memory scramblers in commodity processors without incurring any performance overhead. While the encryption scheme we analyze here cannot prevent bus snooping and memory replay attacks, it is sufficient for preventing any form of cold boot attack.

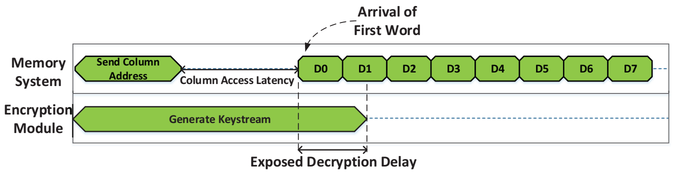


Figure 5: **Minimizing Decryption Overhead.** The key to minimizing memory cipher overheads is to avoid serializing memory access and cryptography and to instead overlap cryptography with memory access. Stream ciphers (*e.g.*, AES CTR) make it is possible to generate the keystream in parallel with accessing memory. If the keystream generation completes within the time required to transfer data from a DRAM row buffer (*i.e.*, the fastest DRAM access), there will be no exposed latency for strongly encrpyted memory. Our analyses show that there are modern crypto engines that are indeed fast enough to have zero exposed latency.

Encryption Schemes: We consider two candidate ciphers to replace memory scramblers: AES and ChaCha (8, 12 and, 20 rounds). AES has been the standard cipher for most applications and hardware vendors already have hardware IP for it, making it an attractive candidate. On the other hand, ChaCha20 [23] is gaining popularity due to its strong security guarantees and higher throughput on systems that do not provide AES hardware acceleration. The fact that a pure software implementation of Chacha runs faster than a software implementation of AES has made it very attractive for mobile devices. In fact, for the past two years, nearly 100% of HTTPS connections between Android versions of Chrome and Google have been using ChaCha20 [24]. Two alternative ciphers with a reduced number of rounds, ChaCha8 and ChaCha12, have also been designed for use in systems that are willing to forgo the extra security margins provided by ChaCha20 in return for reduced computational complexity and further increased throughput [23]. Although there are numerous fast stream ciphers that have been proposed in the past, we do not consider them here as they have not undergone the rigorous public cryptanalysis that AES and ChaCha has endured.

AES-CTR and ChaCha operate as counter-based stream ciphers, permitting us to perform keystream generation without having the corresponding plaintext or ciphertext. Instead of encrypting the block directly, these ciphers encrypt an incrementing counter, which is then XOR'd with the plaintext to produce the ciphertext. This mode of operation is particularly attractive for our application because decryption could proceed in parallel with DRAM access. Before we delve into the hardware design trade-offs, we describe how the ciphers were setup.

- **AES:** We use AES in counter mode, with the physical

[1] As of this writing, there is no publicly available performance data for AMD's upcoming memory encryption implementation

address as a counter, and with a nounce[2] and a key generated at boot time. A memory block in DDR3 and DDR4 is 512-bits, which is four times the size of an AES block. To encrypt a memory block we need to generate four key streams using four different counter values. Since the hardware module can be pipelined, it is possible to generate the four key streams using a single hardware module with only one cycle of delay between encryption/decryption of each 16-byte blocks.

- **ChaCha:** Similar to the above scheme, we use the physical address as a counter, along with a key generated at boot time. In addition to a counter, the ChaCha cipher requires a separate nounce. For this nounce, we also rely on the availability of a boot-time random number generator.

Threat Model and Security Guarantees: The above scheme uses a fixed nounce and counter for repeated writes to a single memory block. However, each memory block is encrypted using a unique nounce or counter. This results in the following guarantees and weaknesses:

- Cold Boot Attacks: Since a unique counter value is used for each memory block, an attacker looking at a single snapshot of memory will see memory blocks encrypted using different keystreams. No memory correlation will exist and decrypting memory without knowledge of the AES key will be intractable.
- Bus-Snooping Attacks: An attacker that is able to monitor the memory bus can observe multiple reads and writes to the same memory block. And since the nounce for a given physical address is fixed, the attacker can acquire multiple blocks encrypted using the same nounce and counter. Consequently, an attacker could replay these recorded blocks without detection, thus, our approach does not protect the system against bus replay attacks. More capable technologies such as Intel's SGX can prevent such attacks at the cost of reduced performance [25], [26].

Minimizing Encryption Overhead: The most straight forward way to encrypt/decrypt bus transactions is to perform the keystream generation when data arrives in the memory controller. The main problem with this approach is that it introduces unacceptable delays on memory reads. Delays on memory writes are tolerable as the CPU can proceed with other tasks while stores are being performed. It is crucial that we reduce decryption delays since memory read latency is one of the major bottlenecks in today's systems.

Multiple works in the past have explored schemes to overlap cryptographic computations with memory reads [15], [20], [27]–[31]. One way to reduce the overhead of decrypting memory reads is to overlap the process of keystream generation with data transfer on the bus. Figure 5 shows the final portion of the memory read process in the DDR protocol. After a row has been read into the row buffer, the memory controller sends column access (CAS) signals. The amount of time it will take for the DRAM module to place the requested columns on the bus is deterministic and fixed for the specific DRAM module.

We leverage the deterministic time window that is available between a DRAM read request and a response from DRAM to hide the overhead incurred by memory encryption. This time window can be used to perform keystream generation, which runs independent of the data for both AES in counter-mode or ChaCha. If the entire keystream generation can be completed within this time window, then the CPU will not experience any delays for implementing fully encrypted memory.

Analyzing the Impact of Full Memory Encryption: To quantify the time window available for key expansion, we looked at the timing characteristics of DDR4 DRAM modules. According to the DDR4 standard there are only 9 allowable column access latencies that manufacturers are allowed to target. All of these standard column access latencies are between 12.5ns and 15.01ns [32]. We use these numbers as a basis for measuring exposed latency due to strong encryption. Implementations of alternative memory standards such as the Hybrid Memory Cube (HMC) have even higher transfer latency in return for higher throughput SerDes links [33].

To evaluate the performance overhead, we must know the keystream generation delay for the ciphers. To assess this delay we ran RTL simulation and synthesis on efficient AES and ChaCha implementations. Our design exploration for AES is based on a modified version of an open-source design [34]. We used the Synopsis Design Compiler to synthesize the designs to a 45nm silicon-on-insulator (SOI) technology library. Since this is a trailing edge technology, the results we generate will be slightly pessimistic compared to what a design might achieve in a newer silicon technology. However, we expect the comparisons we make below with respect to older 45nm CPUs to hold true for newer silicon technology since both the encryption pipeline and the CPUs will scale in a similar manner.

Hardware Design Trade-offs: Depending on different design decisions, the encryption modules can be optimized for latency, throughout, or low-power operation. Here, we detail the different design decisions we made.

Speed vs Area and Power: Both AES and ChaCha apply the same round function multiple times on a block of data. This gives us the option to have a single hardware unit for a round function and time-multiplex it. Such design will result in lower throughput, but also lower power. In addition, high-performance memory controllers can have multiple outstanding requests. For this reason, it is advantageous to chain multiple instances of the hardware units for the round function. In the designs we evaluated, we have dedicated units for each round. These units are then pipelined for increased throughput with multiple outstanding requests.

AES Pipeline Stages: AES rounds can be implemented with lookup tables, and this makes them amenable for faster

[2] A nounce is an input value that is not supposed to be used more than once. A unique nounce is typically generated for every encryption/decryption operation.

Cipher	Maximum Freq.(GHz)	Cycles per 64B	Maximum Pipeline Delay (ns)
AES-128	2.4	13	5.4
AES-256	2.4	17	7.08
ChaCha8	1.96	18	9.18
ChaCha12	1.96	26	13.27
ChaCha20	1.96	42	21.42

Table II: **Cipher Engine Performance (45nm).** This table provides the speed of the five cipher engines analyzed. All implementations were synthesized to a 45nm silicon-on-insulator technology. The latencies presented here do not include potential queuing delays.

designs. The design we used for this evaluation was adapted from [34], and it implements the sub-byte, shift row, and mix column steps as register look ups. The deeply pipelined design in [34] takes 2 cycles per round, and it is capable of running at 2.5 GHz in 45nm silicon, providing a maximum throughput of 40 GB/s. However, we chose to pipeline the design in a way that only takes 1 cycle per round, thereby slightly reducing its maximum clock frequency to 2.4GHz which reduces throughput to 39 GB/s. This slight reduction in throughput enabled us to lower the latency of generating a 16-byte key stream from a counter by about 50%.

ChaCha Pipeline Stages: Implementing a ChaCha quarter round in hardware requires a chain of 32-bit adders and XOR gates. In our design, we broke a quarter round into 2 pipeline stages. This enabled us to clock the design about 2 times faster (at 1.96 GHz) relative to a design where a quarter round is a single pipeline stage. This increased the frequency and resulted in a modest reduction of the latency. As we will outline in our results, this frequency enables the encryption engine to keep up with high-speed buses.

C. Results and Discussion

Cipher Engine Performance: Table II presents the performance characteristics of the synthesized cipher engines. We can see the latency that would be incurred by these cryptographic modules is not acceptable unless it can be hidden by overlapping the key generation with DDR4 DRAM column reads. Since any DDR4 module would take at least 12.5ns for a column access with a row buffer hit, AES-128, AES-256, and ChaCha8 seem like viable alternatives. The numbers also suggest that AES-128 would have lower latency when even compared to ChaCha8. However, there is an advantage to using ChaCha8 under higher bandwidth utilizations. Since AES operates on 16-byte blocks (as opposed to 64-byte blocks in ChaCha), we need to load 4 counters into the pipeline for each 64-byte memory block. This property of AES can become a disadvantage when there are numerous row buffer hits on a single channel (*i.e.*, under high bandwidth utilization).

To analyze the performance of the cipher engines under high bandwidth utilization, we simulated the performance of the modules under different loads. Higher bandwidth utilization occurs when there are multiple row buffer hits across different banks. In the DDR4 standard, even if we might have dozens of banks on a channel, the total number of outstanding CAS commands will ultimately be limited by the contention on the bus. With a fast DDR4 module running at 1.2GHz (DDR4-2400), we can theoretically have up to 18 back-to-back CAS requests, provided that there are enough row buffer hits.

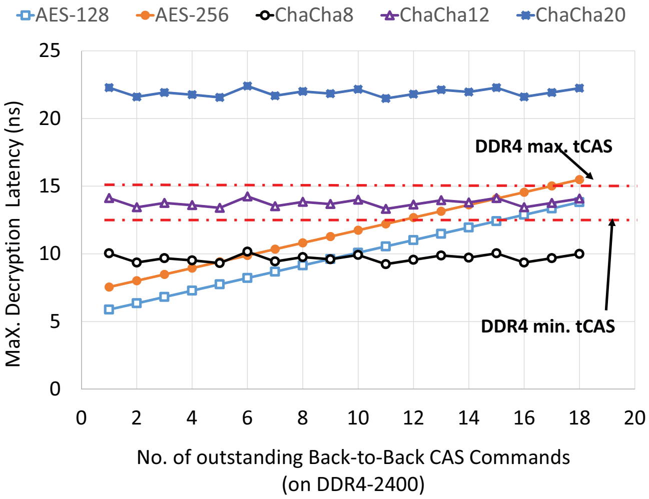


Figure 6: **Decryption Latency of Different Ciphers.** ChaCha8 is able to complete decryption faster than the minimum DDR4 read delay (12.5 ns), thus, there would be no exposed latency on encrypted DRAM reads under all loads. At lower bandwidth utilization (*i.e.*, fewer back-to-back reads with different keys), AES exhibits better performance. However, as the bandwidth utilization approaches its peak, the queuing delay starts slow AES, while ChaCha8 continues to perform well.

Figure 6 graphs the performance of the cipher engines at varying levels of memory bandwidth utilization for a DDR4-2400 module. Note that the standard CAS latencies under DDR4 all lie between 12.5ns and 15.01ns. When the number of outstanding requests is low, AES-128 and AES-256 show superior performance. However, as the number of outstanding requests increase, the queuing delay at the input of the AES modules starts to affect the latency. As mentioned earlier, this results from the need to feed 4 counter/nounce values into the AES pipeline for every column read operation. On the other hand, ChaCha produces a 64-byte keystream from a single counter/nounce. And since this module can be clocked at least as fast as any DDR4 bus, there will be no queuing delays incurred.

The results show that ChaCha8 and AES-128 are the most suitable ciphers for replacing memory scramblers. ChaCha8 is able to complete decryption faster than the minimum DDR4 read delay under all loads. AES-128 would also have zero exposed latency except when subjected to excessive outstanding CAS requests. Even under maximum outstanding back-to-back CAS requests, AES-128 would only have a worst case exposed latency of 1.3ns.

Power and Area Overhead: To understand the power and area overhead of replacing scramblers with strong cipher

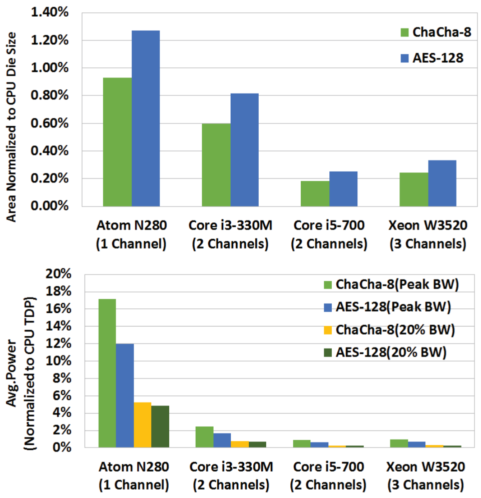


Figure 7: **Power and Area Overhead.** This figure gives estimated area and power overheads for multiple platforms, ranging from a low-end CPU (Atom N280) to a high-end server (Xeon W3520). Area overheads are uniformly low, and power overheads for the larger more capable cores are also low. The Atom CPU overheads grow, but are mitigated for lower channel utilization.

engines, we compare the size and power consumption of ChaCha8 and AES-128 modules against various Intel cores. We perform a technology neutral comparison, as both the cores and cipher engines are implemented in 45nm silicon. Additionally, technology scaling is unlikely to change these results, since both the cores and cipher engines would scale in a similar manner. We make power and area comparisons against 45nm Intel CPUs: the Atom N280 (mobile), Core i3-330M (desktop), Core i5-700 (high-end desktop), and Xeon W3520 (server) CPUs. We used the power profiles and die size values stated on their respective product sheets. The power and area results are presented in Figure 7. We assume that there is one encryption module per-channel for each of the comparisons. As a result, we multiplied the power and area numbers of a single encryption module by the number of channels in the system.

We used the Synopsis Design Compiler for power (static and dynamic) and area estimation. For estimating the dynamic power, we used signal activity factors under full bandwidth utilization, where back-to-back CAS requests are generated whenever the bus is free. As previous work [35] has shown that most workloads utilize only a fraction of DRAM bandwidth, we also present power overheads for 20% utilization by scaling down the dynamic power to 20% of the maximum dynamic power. The analysis in [35] shows that even data intensive applications such as media streaming only use up to 15% of DRAM bandwidth which makes our estimates at 20% bandwidth utilization conservative.

Clearly, the overall power and area overheads for strong encryption are very low. In all cases, the area overheads are about or below 1%, with the expected slightly higher overheads on the small Atom CPU. The power overheads are all below 3%, except for the single core Atom CPU, which experiences up to a 17% power increase under full bandwidth utilization. This is to be expected due to the greatly increased energy efficiency of the Atom CPU. Under more realistic workloads, however, the power overhead of the Atom CPU is estimated to be below 6%.

For low-power mobile devices, more energy-efficient memory encryption can be achieved by using cipher engines that have much lower performance than what we proposed here. Such trade-off is possible as mobile-CPUs are not likely to produce a large number of back-to-back CAS requests as server-grade CPUs and co-processors can potentially do.

Key Idea 2: Memory scramblers can be replaced with strong stream ciphers such as ChaCha8. For such low overhead ciphers, the process of keystream generation, which is independent of the data being encrypted, can be fully overlapped with DRAM row buffer access – thereby completely hiding the overhead of data decryption during memory reads.

V. Conclusion

With the introduction of memory scramblers in modern processors, cold boot attacks have become more challenging, as attackers must first descramble the contents of DRAM. In this work, we demonstrated that the weak data obfuscation afforded by scramblers can be readily overcome. We develop and demonstrate a straightforward means to descramble DDR4 DRAM connected to an Intel Skylake CPU by exploiting the data correlations that are created due to the reuse of a limited number of scrambler keys. We presented a cold boot attack that is able to extract AES keys (including VeraCrypt/TrueCrypt master keys) from scrambled memory. Finally, we show hardware encryption performance results that suggest that memory scramblers could be readily replaced with strong stream ciphers without incurring any performance overhead. We show that ChaCha8 can fully overlap decryption with the row buffer reads in a DDR4 DRAM module, leaving no exposed latency for strongly encrypted DRAM. Similarly, we show that the power overheads for implementing a strongly encrypted DRAM are quite low. Given the increasing size of memories and the introduction of non-volatility, memories are prone to holding more secrets for longer periods of time. As such, it is becoming increasingly important to protect the contents of system RAM. If hardware vendors adopt the low-overhead strong stream ciphers as laid out in this paper, we can effectively defend systems against future cold boot attacks.

VI. Acknowledgments

The authors would like to thank Patipan Prasertsom, Doowon Lee, Zelalem Aweke and the reviewers, whose insights improved this work. This work was supported in part by C-FAR, one of the six STARnet centers, sponsored by MARCO and DARPA.

References

[1] S. Skorobogatov, “Low Temperature Data Remanence in Static RAM,” *University of Cambridge Computer Laborary Technical Report*, vol. 536, 2002.

[2] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley Publishing, 2008.

[3] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest We Remember: Cold-Boot Attacks on Encryption Keys,” *Communications of the ACM*, vol. 52, no. 5, 2009.

[4] T. Müller and M. Spreitzenbarth, “FROST: Forensic Recovery of Scrambled Telephones,” in *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, 2013.

[5] P. Mosalikanti, C. Mozak, and N. Kurd, “High Performance DDR architecture in Intel® Core™ Processors Using 32nm CMOS High-K Metal-Gate Process,” in *Proceedings of the IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2011.

[6] Intel Corporation, *6th Generation Intel® Processor Datasheet for S-Platforms*, 2015.

[7] M. Gruhn and T. Muller, “On the Practicability of Cold Boot Attacks,” in *Proceedings of the Eighth IEEE International Conference on Availability, Reliability and Security (ARES)*, 2013.

[8] I. Skochinsky. Secrets of Intel Management Engine. Accessed: 2016-02-17. [Online]. Available: http://www.slideshare.net/codeblue_jp/igor-skochinsky-enpub

[9] J. Bauer, M. Gruhn, and F. C. Freiling, “Lest We Forget: Cold-Boot Attacks on Scrambled DDR3 Memory,” *Digital Investigation*, vol. 16, 2016.

[10] S. Lindenlauf, H. Hofken, and M. Schuba, “Cold Boot Attacks on DDR2 and DDR3 SDRAM,” in *Proceedings of the 10th IEEE International Conference on Availability, Reliability and Security (ARES)*, 2015.

[11] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals*. Pearson Education, 2012.

[12] P. Simmons, “Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption,” in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[13] T. Müller, F. C. Freiling, and A. Dewald, “TRESOR Runs Encryption Securely Outside RAM,” in *Proceedings of the 20th USENIX Conference on Security*, 2011.

[14] M. Henson and S. Taylor, “Memory Encryption: A Survey of Existing Techniques,” *ACM Computing Surveys (CSUR)*, 2013.

[15] J. Yang, L. Gao, and Y. Zhang, “Improving Memory Encryption Performance in Secure Processors,” *IEEE Transactions on Computers*, 2005.

[16] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell *et al.*, “SCONE: Secure Linux Containers with Intel SGX,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[17] D. Kaplan, J. Powell, and T. Woller, “AMD Memory Encryption Whitepaper.”

[18] Intel Corporation, *Intel® Atom™ Processor S1200 Product Family for Microserver*, 2012.

[19] JEDEC Solid State Technology Association and others, “DDR4 SDRAM SO-DIMM Design Specification,” *JEDEC Standard No. 21C*, 2016.

[20] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, “Phantom: Practical Oblivious Computation in a Secure Processor,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[21] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path ORAM: An Extremely Simple Oblivious RAM Protocol,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[22] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, “Tec-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007.

[23] D. J. Bernstein, “ChaCha, A Variant of Salsa20,” in *Workshop Record of SASC*, vol. 8, 2008.

[24] E. Bursztein. Speeding up and Strengthening HTTPS Connections for Chrome on Android. Accessed: 2016-02-17. [Online]. Available: https://googleonlinesecurity.blogspot.com/2014/04/speeding-up-and-strengthening-https.html

[25] S. Gueron, “Intel’s SGX Memory Encryption Engine,” *Proceedings of the Real World Cryptography Conference*, 2016.

[26] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, “Innovative Technology for CPU Based Attestation and Sealing,” in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (WASP)*, vol. 13, 2013.

[27] V. Young, P. J. Nair, and M. K. Qureshi, “DEUCE: Write-Efficient Encryption for Non-Volatile Memories,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[28] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, “Efficient Memory Integrity Verification and Encryption for Secure Processors,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2003.

[29] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, “Improving Cost, Performance, and Security of Memory Encryption and Authentication,” in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, 2006.

[30] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas, “Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[31] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS and Performance Friendly,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2007.

[32] JEDEC Solid State Technology Association and others, “JEDEC Standard: DDR4 SDRAM,” *JESD79-4*, 2012.

[33] A. Sodani, “Intel Xeon Phi Processor “Knights Landing” Architectural Overview.”

[34] H. Hsing. (2013) AES Core. Accessed: 2016-02-17. [Online]. Available: http://opencores.org/project,tiny_aes

[35] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.