2018 IEEE 36th International Conference on Computer Design

NVCool: When Non-Volatile Caches Meet Cold Boot Attacks*

Xiang Pan†, Anys Bacha‡, Spencer Rudolph†, Li Zhou†, Yinqian Zhang†, Radu Teodorescu†
†The Ohio State University
{panxi, rudolph, zholi, yinqian, teodores}@cse.ohio-state.edu
‡University of Michigan
bacha@umich.edu

***Abstract*—Non-volatile memories (NVMs) are expected to replace traditional DRAM and SRAM for both off-chip and on-chip storage. It is therefore crucial to understand their security vulnerabilities before they are deployed widely. This paper shows that NVM caches are vulnerable to so-called "cold boot" attacks, which involve physical access to the processor's cache. SRAM caches have generally been assumed invulnerable to cold boot attacks, because SRAM data is only persistent for a few milliseconds even at cold temperatures.**

Our study explores cold boot attacks on NVM caches and defenses against them. In particular, this paper demonstrates that hard disk encryption keys can be extracted from the NVM cache in multiple attack scenarios. We demonstrate a reproducible attack with very high probability of success. This paper also proposes an effective software-based countermeasure that can completely eliminate the vulnerability of NVM caches to cold boot attacks with a reasonable performance overhead.

I. Introduction

Non-volatile memory (NVM) such as Spin-Transfer Torque Random Access Memory (STT-RAM), Phase Change Memory (PCM), and Resistive Random Access Memory (ReRAM), is a promising candidate for replacing traditional DRAM and SRAM memories for both off-chip and on-chip storage [1]. NVMs in general have several desirable characteristics including non-volatility, high density, better scalability at small feature sizes, and low leakage power [2]–[4]. Prior work has examined the performance, energy, and reliability implications of NVM register files, caches, and main memories [2]–[8].

The semiconductor industry is investing heavily in NVM technologies and companies such as Everspin [9] and Crossbar [10] are focused exclusively on NVM technologies and have produced NVM chips that are being sold today. A joint effort from Intel and Micron has yielded 3D XPoint [11], a new generation of NVM devices with very low access latency and high endurance, expected to come to the market this year. Hewlett Packard Enterprise's ongoing "The Machine" project [12] is also set to release computers equipped with memristor technology (also known as ReRAM) as part of enabling highly scalable memory subsystems. The industry expects non-volatile memory to replace DRAM off-chip storage in the near future, and SRAM on-chip storage in the medium term.

This work was supported in part by the National Science Foundation under grants CCF-1253933 and CCF-1629392, and The Ohio Supercomputer Center.

Despite all their expected benefits, non-volatile memories will also introduce new security vulnerabilities as data stored in these memories will persist even after being powered-off. In particular, non-volatile memory is especially vulnerable to "cold boot" attacks. Cold boot attacks, as first proposed by Halderman et al. [13], use a cooling agent to lower the temperature of DRAM chips before physically removing them from the targeted system. Electrical characteristics of DRAM capacitors at very low temperatures cause data to persist in the chips for a few minutes even in the absence of power. This allows the attacker to plug the chips into a different machine and scan the memory image in an attempt to extract secret information. When the memory is implemented using NVM, the cold boot attacks become much simpler and more likely to succeed, because data persists through power cycles.

To protect against cold boot attacks on main memory, one approach is to encrypt sensitive data [14]–[21]. Another approach is to keep secret keys stored in SRAM-based CPU registers, caches, and other internal storage during system execution [22]–[31]. The rationale behind this design philosophy is that cold boot attacks against on-chip SRAM structures are deemed to be extremely difficult. This is because SRAM data persistence at cold temperatures is limited to a few milliseconds [32].

The security implications of implementing the main memory and microprocessor caches with NVM have received little attention. While memory encryption schemes are feasible, cache encryption is not practical due to low access latency requirements. Cold boot attacks on *unencrypted NVM caches* will be a serious concern in practice, especially given the ubiquity of smart mobile and Internet of Things (IoT) devices, which are more exposed to physical tampering — a typical setup for cold boot attacks.

This work examines the security vulnerabilities of microprocessors with NVM caches. In particular, we show that encryption keys can be retrieved from NVM caches if an attacker gains physical access to the device. Since removing the processor from a system no longer erases on-chip memory content, sensitive information can be leaked. This paper demonstrates that AES disk encryption keys can be identified in the NVM caches of a simulated ARM-based system running the Ubuntu Linux OS.

We have examined multiple attack scenarios to evaluate the

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00072

probability of a successful attack depending on the system activity, attack timing, and methodology. In order to search for AES keys in cache images, we adopted the key search algorithm presented in [13] for main memories, and made the necessary modifications to target caches which cover non-contiguous subsets of the memory space. We find that the probability of identifying an intact AES key if the processor is stopped at any random point during execution ranges between 5% and 100%, depending on the workload and cache size. We also demonstrate a reproducible attack with 100% probability of success for the system we study.

To counter such threats, this paper proposes an effective software-based countermeasure. We patch the Linux kernel to allocate sensitive information into designated memory pages that we mark as uncacheable in their page table entries (PTEs). This way secret information will never be loaded into vulnerable NVM caches but only stored in main memory and/or hard disk, which can be encrypted with a reasonable performance cost. The performance overhead of this countermeasure ranges between 2% and 45% depending on the hardware configuration.

Overall, this paper makes the following main contributions:

- The first work to examine cold boot attacks on non-volatile caches.
- Two types of cold boot attacks have been performed and shown to be effective on non-volatile caches.
- A software-based countermeasure has been developed and proven to be effective.
- An algorithm for identifying AES keys in cache images has been implemented.

The rest of this paper is organized as follows: Section II provides background information and discusses related work. Section III explains the threat model. Section IV illustrates our cache-based AES key search algorithm. Section V describes our experimental methodology. Section VI presents and evaluates two types of cold boot attacks. Section VII presents countermeasures and evaluates their effectiveness and performance overhead. Finally, section VIII concludes.

II. Background and Related Work

In this section we provide some background information relevant to our study and discuss related work in the context of cold boot attacks.

A. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [33] is a symmetric block cipher that encrypts or decrypts a block of 16 bytes of data at a time. Both encryption and decryption use the same secret key. The commonly used AES key size is either 128-bit (AES-128), 192-bit (AES-192), or 256-bit (AES-256). Before performing any encryption/decryption operations, the secret key must be expanded to a key schedule consisting of individual subkeys that will be used for different internal rounds of the AES (Rijndael) algorithm. The key expansion process is shown in Figure 1. The key schedule starts with the original secret key, which is treated as the initial subkey. The

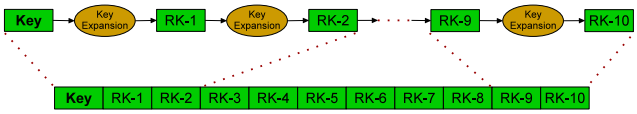

Fig. 1: AES-128 key schedule with details on key expansion.

following subkeys (also known as round keys) are computed from the previously generated subkeys using publicly known functions such as `Rot-Word`, `Sub-Word`, `Rcon`, `EK`, and `K` defined in the key expansion algorithm [34]. In each round of the key generation, the same sequence of functions will be executed. Each newly generated subkey will have the same size, e.g. 16 bytes in AES-128. This process repeats a certain number of rounds (e.g. 10 rounds in AES-128) until the expanded key is completely generated. Each subkey from the expanded key will then be used in separate rounds of AES encryption or decryption algorithms.

B. ARM's Cryptographic Acceleration

Many of today's processors include vectorization support in the form of Single Instruction Multiple Data (SIMD) engines. In ARM processors, the SIMD engine, called NEON, consists of 32 128-bit registers (v0 - v31). Each NEON register can conveniently load a 128-bit AES key or a full round of the AES key schedule into a single register obviating the need for multiple accesses to the cache or the memory subsystem. In addition, ARMv8 introduces cryptographic extensions that include new instructions that can be used in conjunction with NEON for AES, SHA1, and SHA2-256 algorithms. In the case of AES, the available instructions are: `AESD` for single round decryption, `AESE` for single round encryption, `AESIMC` for inverse mix columns, and `VMULL` for polynomial multiply long. In this paper, we explore the use of the NEON engine, in addition to the ARMv8 cryptographic extensions.

C. Cold Boot Attacks and Defenses

The idea of cold boot attacks in modern systems was first explored by Halderman et al. [13]. Their work consisted of extracting disk encryption keys using information present in main memory (DRAM) images preserved from a laptop. The idea of this type of attack builds on the premise that under low temperature conditions, DRAM chips preserve their content for extended time durations. The attack also relies on the fact that AES keys can be inferred by examining relationships between subkeys that involve computing the hamming distance information. The AES key search algorithm has been proposed in [13] as well. Muller et al. [35] later expanded cold boot attacks to mobile devices. When volatile memories such as SRAM and DRAM are replaced by non-volatile ones (e.g. STT-RAM, PCM, and ReRAM) in future computers, cold boot attacks will become much easier to perform since data will be indefinitely preserved after cutting off power supply for several years without the need for any special techniques such as cooling. Our work is the first work to study cold boot attacks in the context of non-volatile caches.

Prior work has proposed encrypting memory data in order to prevent attackers from easily extracting secret information from main memory [14]–[21]. Although this approach is effective for main memory, encryption techniques are challenging to apply to caches because of their large performance overhead. Other researchers, proposed storing secret keys away from main memory in CPU registers, caches, and other internal storage during system execution [22]–[28], [30], [31]. However these proposed approaches have not considered the vulnerability of data stored in CPU caches, especially since keys stored in CPU registers and other internal storage can still be fetched into caches during execution [23]–[25], [28], [30].

III. Threat Model

The threat model assumed in this study is consistent with prior work on "cold boot" attacks. In particular, we assume the attacker gains physical access to the target device (e.g. an IoT device, a smartphone, a laptop, or a desktop computer). Further, the attacker is assumed to have the ability to extract the microprocessor or system motherboard from the device and install them into a debugging platform, which allows cache data to be accessed. In practice, such a platform is not hard to obtain. Many microprocessor manufacturers offer debugging and development platforms that allow a variety of access functions, including functionality to retrieve the cache content.

For example, for the ARM platform, the manufacturer offers the DS-5 development software [36] and associated hardware DSTREAM Debug and Trace unit [37]. These tools enable debugging and introspection into ARM processor-based hardware. The attacked microprocessor can be plugged into a development board such as the Juno ARM Development Platform [38] either directly or through the JTAG debugging interface. In the DS-5 software, the *Cache Data View* can be used to examine the contents of all levels of caches and TLBs. Information such as cache tags, flags, and data associated with each cache line, as well as the index of each cache set, can be read and then exported to a file for further processing.

IV. Cache-Aware AES Key Search

In this paper, we study the security of NVM-based microprocessor caches specifically by demonstrating AES key extraction attacks under the aforementioned threat model.

A. Technical Challenges

An algorithm for identifying AES keys in a main memory image has been presented by Halderman et al. in their seminal work on cold boot attacks [13]. Its application to caches, however, is not straightforward. Particularly, the AES key search algorithm in Halderman et al. [13] assumes that a *complete* AES key schedule is stored in a *physically-contiguous* memory region. This is a relatively safe assumption in their case since the size of memory pages on modern computers are at least 4KB and a complete AES key schedule is 176 bytes (128-bit key/AES-128) to 240 bytes (256-bit key/AES-256).

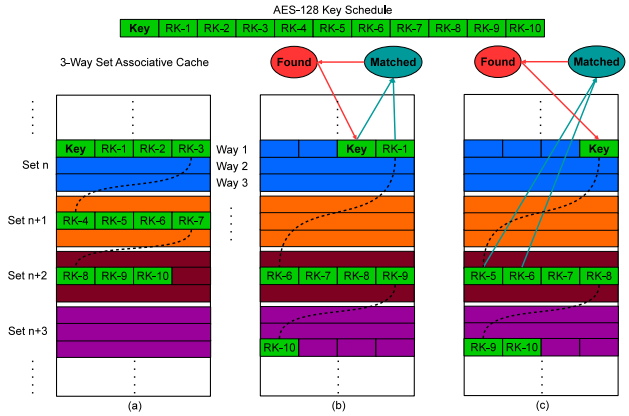

Fig. 2: Cache view with (a) complete and (b, c) incomplete AES key schedules stored in disjoint 64-byte lines.

The algorithm proposed by Halderman et al. can, therefore, simply scan the entire memory image sequentially to search for potential AES keys [13]. However, neither the *completeness* of the key schedule nor the *contiguity* of the memory space can be assumed in the case of caches.

Non-contiguous memory space. Caches only capture a small non-contiguous subset of the memory space. Since cache lines are typically only 32-128 bytes, data that was originally stored in physically-contiguous memory is not necessarily stored in contiguous cache regions. Therefore, the logically sequential AES key schedules, typically 176 to 240 bytes, can be separated into multiple physically disjoint cache lines as shown in Figure 2(a).

Incomplete key schedules. Another relevant cache property is that data stored in the cache is subject to frequent replacements. Parts of a complete AES key schedule can be missing from the cache, which makes our key search more difficult to conduct. Examples of these situations are shown in Figure 2(b) and 2(c). Particularly, in Figure 2(b), the cache line that holds the `RK-2`, `RK-3`, `RK-4`, and `RK-5` has been evicted from the cache.

B. Search Algorithm Design

To address these issues our algorithm will first reconstruct the cache image by sorting cache lines by their physical addresses that we extract from the cache tags and indexes, and then feed the reconstructed cache image to the AES key search function. In this way the logically contiguous data will still be contiguous in our reconstructed cache image regardless of the cache architecture.

QuickSearch. Our key search algorithm runs through each key schedule candidate (all cache words) and first attempts to validate the first round of the key expansion (16 bytes). If there is a match between the first round expansion of the candidate key and the data stored in the cache, the candidate key is validated. As long as the key itself followed by one round (16 bytes) of the expanded key exists in the cache, our algorithm

can successfully detect the key as shown in Figure 2(b). We call this variant of the key search algorithm, *QuickSearch*.

DeepSearch. The AES key schedule is stored on multiple cache lines since it is larger (at least 176 bytes for AES-128 mode) than the typical cache block (32-128 bytes). Cache evictions can displace parts of the AES key schedule from the cache, including the first round of the key expansion, which our *QuickSearch* algorithm uses to validate the key. These cases are rare since they require the memory alignment to be such that the encryption key falls at the end of a cache line and the first round of the key expansion is on a different line. To deal with these cases we designed a more in-depth algorithm (which we call *DeepSearch*) that considers multiple rounds of the key expansion. In this implementation, as long as the key itself is inside the cache and there exist two consecutive rounds of expanded keys, our algorithm can find the key as shown in Figure 2(c). The downsides of *DeepSearch* is that it runs considerably slower than *QuickSearch* and the search has some false positives.

C. Implementation-Specific Considerations

We demonstrate the AES key extraction attack against `dm-crypt`, a cryptographic module that is used in mainstream Linux kernels. As will be explained in Section V, the specific target of our demonstrated attacks is the disk encryption/decryption application, LUKS (Linux Unified Key Setup), of the Ubuntu OS, which by default invokes the `dm-crypt` kernel module for disk encryption/decryption using AES-XTS mode [39].

In the AES implementation of `dm-crypt` the decryption key schedule is different from the encryption one. We illustrate the key schedule for the decryption process in Figure 3(a). The decryption key schedule is first reversed in rounds from the encryption key schedule. An inverse mix column operation is then applied to rounds 1 through 9 of the key schedule. As a result, we need to perform searches for encryption keys and decryption keys separately. Specifically, before searching for decryption keys we first convert the candidate schedules back to the encryption key schedule format.

Another artifact that affects our key search algorithm is specific to little-endian machines, which store the least significant byte in the lowest address. `dm-crypt` adopts an implementation which stores the AES key schedules as an array of words (e.g. 4 bytes) instead of bytes. This leads to a mismatch in the representation on little-endian architectures, as shown in the first line of Figure 3(b). Our search algorithm takes into account this artifact and converts the little-endian representation to big-endian before conducting the key search.

V. Experimental Methodology

We used a full system simulator to conduct our experiments since microprocessors with NVM caches are not currently available in commercial systems. Specifically, we modeled an 8-core ARMv8-based processor using the gem5 simulator [40]. Cold boot attacks have been demonstrated on both x86 and ARM architectures in the past. We used the ARMv8

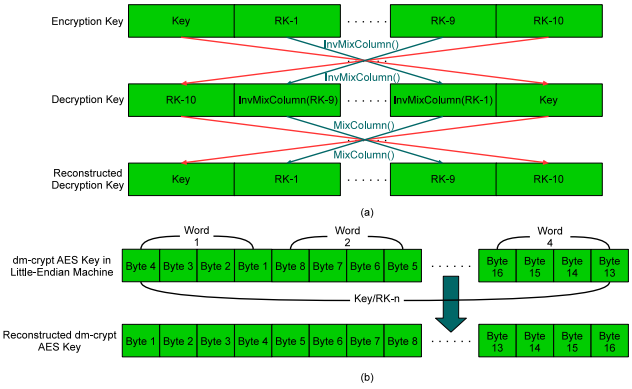

Fig. 3: Details on AES implementation-dependent modifications to the key search algorithm.

Hardware Configuration	
Cores	8 (out-of-order)
ISA	ARMv8 (64-bit)
Frequency	3GHz
IL1/DL1 Size	32KB
IL1/DL1 Block Size	64B
IL1/DL1 Associativity	8-way
IL1/DL1 Latency	2 cycles
Coherence Protocol	MESI
L2 Size	2, 4, 8 (default), and 128MB
L2 Block Size	64B
L2 Associativity	16-way
L2 Latency	20 cycles
Memory Type	DDR3-1600 SDRAM
Memory Size	2GB
Memory Page Size	4KB
Memory Latency	300 cycles
Disk Type	Solid-State Disk (SSD)
Disk Latency	150us

TABLE I: Summary of hardware configurations.

architecture for its broader adoption in mobile devices which are particularly vulnerable to the physical access required by cold boot attacks. Our results should be generally applicable to other microprocessors.

We simulated a 2-level cache hierarchy with private L1 instruction and data caches for each core and a shared inclusive L2 cache as the last level cache (LLC). Our cache configuration parameters are in line with the ones used in many modern computer systems. Since our key search algorithm focuses on the LLC, we experimented with different LLC sizes from 2MB to 128MB. We examine a wide range of LLC sizes from small (2MB) to very large (128MB), with most experiments conducted using an 8MB LLC. The main configuration parameters of our simulated system are summarized in Table I.

The system is configured to run Ubuntu 14.04 Trusty 64-bit operating system. We installed the cryptsetup application - LUKS (Linux Unified Key Setup) in the Ubuntu OS and used it with the `dm-crypt` module in Linux kernel to encrypt a 4GB partition of a simulated hard drive. The disk encryption/decryption algorithm we configured for LUKS was AES-XTS [39] with 128-bit keys. The XTS format is currently the standard form for Linux since ECB/CBC format has known

Mixed Benchmark Groups		
mixC	compute-bound	*calculix, dealII, gamess, gromacs, h264ref, namd, perlbench, povray*
mixM	memory-bound	*astar, cactusADM, GemsFDTD, lbm, mcf, milc, omnetpp, soplex*
mixCM	compute/ memory	*dealII, gamess, namd, perlbench, astar, cactusADM, lbm, milc*

TABLE II: Detailed makeup of the mixed benchmark groups.

NVCool Experiments	
NoNEON	System without ARM's cryptographic acceleration support
NEON	System with ARM's cryptographic acceleration support
STAvg	Geometric mean of single-threaded benchmarks from SPEC CPU2006

TABLE III: Experiment names and short description.

security flaws [41]. One detail to note here is that AES-XTS uses a dual encryption/decryption method which requires two different AES keys to perform encryption/decryption operations [33]. Our experiments take this into account and only consider the key search a success when both keys are found.

We ran the SPEC CPU2006 benchmark suite with both binaries and data stored in the encrypted hard drive to simulate applications that run on the target system. SPEC CPU2006 includes integer and floating-point single-threaded benchmarks, with a mix of computation-bound and memory-bound applications [42].

To keep the simulation time reasonable, we use the checkpoint functionality provided by gem5 [40] to bypass the OS boot-up phase and ran each benchmark with up to 1 billion instructions. For experiments which require periodically taking LLC image snapshots we use a sampling interval of 1 million instructions. To further test our attack scenario and countermeasure approach in a multi-programmed/multi-threaded environment, we also ran several groups of mixed benchmarks from SPEC CPU2006 - *mixC*, *mixM*, and *mixCM*. As detailed in Table II, *mixC* contains 8 computation-bound benchmarks, *mixM* contains 8 memory-bound benchmarks, and *mixCM* contains 4 benchmarks from *mixC* and another 4 benchmarks from *mixM*.

VI. Vulnerability Analysis

We examine the probability of successfully retrieving disk encryption keys from a processor's last level cache under two attack scenarios.

A. Random Information Harvesting

We first investigate an attack scenario in which the attacker gains access to the target machine and disconnects the processor from the power supply at an arbitrary point during the execution. We make no assumptions that the attacker has a way to force a certain code sequence to execute. This is typical, for instance, when a defective device that stores sensitive data is discarded without proper security measures. Another example is when an attacker steals a device and physically disconnects its battery or power supply before removing the processor.

To study the probability of success for such an attack we take periodic snapshots of the LLC, at 1 million instruction intervals. We then run the *QuickSearch* key search algorithm on each cache snapshot. Figure 4 shows the probability of finding the AES keys in the 8MB last level cache for different benchmarks. We examine systems with and without ARM's cryptographic acceleration (NEON) support. For easy reference, Table III summarizes the labels we use for different experiments.

To better analyze results we classify the SPEC CPU2006 benchmarks into two categories — compute-intensive and memory-intensive. We can see from the results in Figure 4 that when running compute-bound benchmarks the probability of finding AES keys in the cache is higher than when running memory bound benchmarks. On average, there is a 76% probability of finding AES keys in the system without NEON support when running compute-bound benchmarks and a 26% probability when running memory-bound benchmarks.

When the system is configured with NEON support, the probability of finding the key in the cache drops for both classes of benchmarks to 41% and 14% respectively. This is because the NEON technology stores encryption keys in vector registers that are large enough to hold the entire key schedule. These registers are also infrequently used for other functions which means they don't have to be spilled to memory (and cache). As a result, in processors with NEON support the encryption key is read from memory much less frequently, leading to better resilience to this type of attack. A typical case is seen in *perlbench* with 96% for NoNEON and 49% for NEON. However there are also exceptions as seen in *povray* (100% for both systems) and *gobmk* (97% for NoNEON and 4% for NEON). On average, the probability of finding the key in this random attack is 40% for the system without NEON support and 22% for the system with NEON support.

There are two principal factors that affect the probability the encryption key is found in the cache at random points during execution. The first is how recent the last disk transaction was, since encrypted disk access requires the AES key to be brought into the cache. The second factor is the cache miss rate, since the higher the churn of the data stored in the cache the sooner an unused key will be evicted. Computation bound benchmarks in general have a smaller memory footprint so their cache miss rates are lower, allowing keys to reside in the cache for longer. Memory bound benchmarks, on the other hand, have a larger memory footprint associated with a higher cache miss rate, therefore evicting keys more frequently.

Figure 5 illustrates these effects for selected benchmarks running on systems without NEON support, showing for each cache snapshot over time whether the key was found or not (1 or 0). The figure also shows the cumulative miss rate of the LLC over the same time interval.

Benchmark *dealII* shown in Figure 5a is a good illustration of the behavior of a compute-bound application. The disk

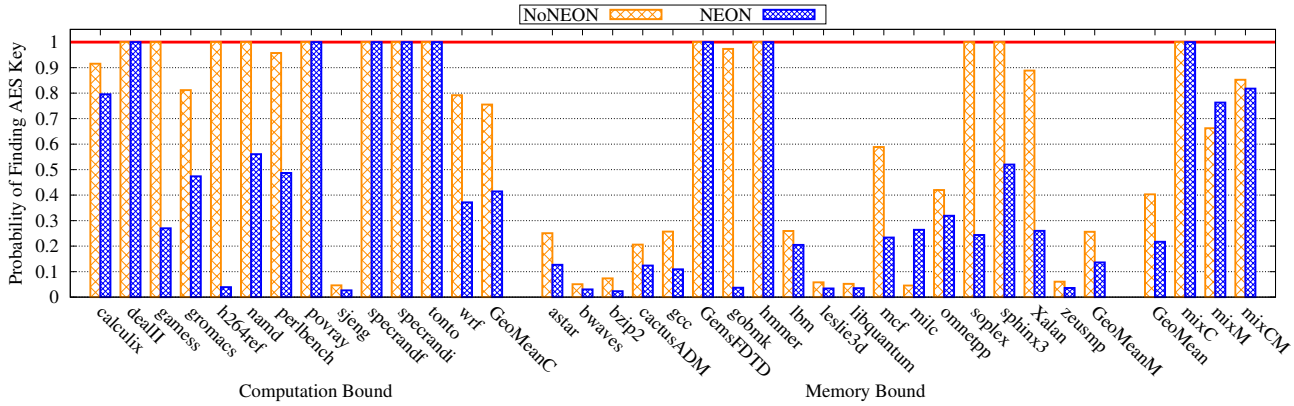

Fig. 4: Probability of finding AES keys in a system with 8MB LLC, as a function of workload.

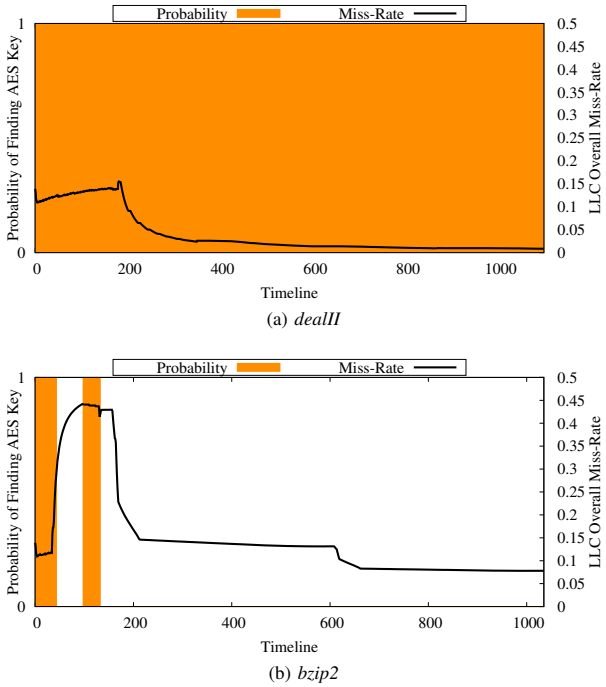

Fig. 5: AES key search trace showing the outcome of the key search and the cumulative LLC miss rate information for (a) *dealII* and (b) *bzip2* benchmarks running on systems without ARM's cryptographic acceleration support (NoNEON).

encryption key is brought into the cache early in the execution as the application accesses the disk to read input data. The miss rate is low throughout the execution of the application which means the key is never evicted and the probability of finding the key while running this application is 100%.

Figure 5b shows the behavior of a memory bound application, *bzip2*. Keys are brought into the cache early in the execution and remain in the cache for a period of time while the miss rate is low. The miss rate, however, spikes as the application begins processing large data blocks for compression. This evicts the key from the cache. A disk operation causes the key to be brought into the cache again, but the consistently high miss rate causes the key to be evicted shortly after that. Even though later in the execution the cache miss rate drops, the lack of disk accesses keep the keys away from the cache for the rest of this run. Note that for clarity we only show a fraction of the total execution.

We also collect results for multi-program mixed workloads to increase system and disk utilization and cache contention. The results are included in Figure 4 as *mixC*, *mixM*, and *mixCM*. The benchmark applications included in each mix are listed in Table II. As expected, when the system is fully utilized, with one application running on each core (for a total of 8), the probability of finding the key increases. This is because each application accesses the disk and those accesses occur at different times, causing the encryption key to be read more frequently. The compute bound *mixC* shows 100% probability of finding the key for both systems (with and without NEON support).

While a system with high utilization is clearly more vulnerable, a mitigating factor is that cache contention is also higher when many threads are running. As a result, cache miss rates are also higher and the key may be evicted more frequently. This is apparent when we examine the memory-bound *mixM* workload which shows a 66% success rate without NEON support and 76% with NEON support. Even with the higher miss rate, the fully-loaded system is clearly more vulnerable than lightly-loaded system, as seen in the single-threaded experiments. When a mix of both compute and memory bound applications is used (*mixCM*) the probability of finding the key is 85% for NoNEON and 82% for NEON.

We also note that the NEON-accelerated system is almost as vulnerable as the system without hardware acceleration when the system is running the mix workloads. This is likely caused by the more frequent spills of the vector registers when context switching between the kernel thread running the encryption/decryption process and the user threads. Spilling

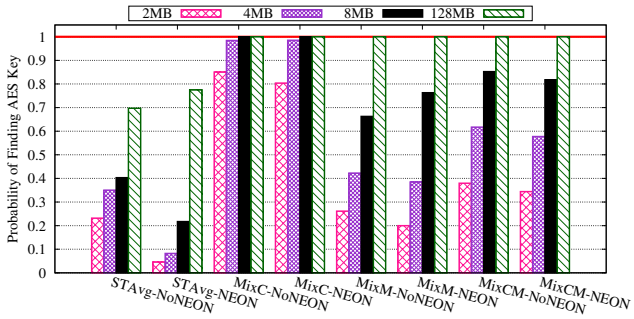

Fig. 6: Overall probability of finding AES keys in LLCs of various sizes.

the vector registers holding the encryption key increases reads and writes of the key to and from memory, exposing it to the cache more frequently.

Figure 6 shows the overall probability of finding AES keys in systems with different LLC sizes. As expected, larger caches increase the system vulnerability to this type of attack. We can see that as cache size increases the probability of success also increases across all the benchmarks. With 2MB cache the average probability of finding AES keys is from 4.6% to 85% depending on the system and application; for a 128MB LLC the probability of a successful attack ranges from 70% to 100%. Increasing the cache size reduces capacity misses therefore increasing the fraction of time the key spends in the cache.

B. Targeted Power-off Attack

The second attack scenario we consider is one in which the attacker is able to trigger a graceful or forced power-off sequence before physically removing the processor. In this attack scenario, the attacker aims to use the power-off sequence to ensure the disk encryption keys are brought to the cache. Since the power-off sequence involves unmounting the disk, this results in a series of encryption/decryption transactions that will bring the encryption key into the cache. During the system shutdown process, the attacker can stop the execution at any time to search for secret keys or simply wait until the device is completely powered off to examine cache images for secret keys. The goal of this attack is to find a reproducible and reliable way to obtain the encryption key from a compromised system.

Figure 7 shows the sequence of operations executed after running the `poweroff` command. There are two operations in the power-off sequence (highlighted in green) that will bring disk encryption keys to the cache. The first (operation 2) is the operating system asking all remaining processes to terminate. In this operation the process in charge of disk encryption will be terminated. This will invoke the `sync` system call to flush data from the page cache to the encrypted disk which requires reading the AES keys. Before the system is actually powered off, all filesystems must be unmounted as shown in step 5.

Mode	Command	Keys exist in cache after power-off?			
		2MB	4MB	8MB	128MB
Normal Power-off	poweroff (-p)	N	N	Y	Y
Forced Power-off	poweroff -f	Y	Y	Y	Y

TABLE IV: Summary of targeted power-off attack results.

```
root@aarch64-gem5:/# poweroff
Session terminated, terminating shell...exit
 ...terminated.
 * Stopping rsync daemon rsync                          [ OK ]
       // 1
 * Asking all remaining processes to terminate...       [ OK ]
       // 2
 * All processes ended within 1 seconds...              [ OK ]
       // 3
 * Deactivating swap...                                 [ OK ]
       // 4
 * Unmounting local filesystems...                      [ OK ]
       // 5
 * Stopping early crypto disks...                       [ OK ]
       // 6
 * Will now halt // 7
[  604.955626] reboot: System halted
```

Fig. 7: The sequence of events triggered by the `poweroff` command.

The encryption keys are used in unmounting the encrypted disk drive and they will again appear in the cache.

We experimented with two power-off methods in the evaluation - normal and forced. We examine the probability of successfully identifying the key under the two scenarios. Table IV summarizes the results of our power-off attacks on various LLC sizes.

We can see from the results that starting from an LLC size of 8MB keys will remain in the cache no matter which power-off method is used. For the smaller cache sizes like 2MB or 4MB, after keys are brought into the cache, other operations which don't involve encryption disk accesses, may evict the AES keys. Therefore we won't see the keys after the system is powered off. However for larger caches with 8MB or 128MB the keys will stay in the cache after system shutdown.

Forced power-off is different from normal power-off in that it doesn't power-off the system in a graceful way. This means forced power-off will only perform the action of powering off the system. However in order to power off the system the local filesystems are still going to be unmounted to prevent data loss. In this process the keys will be brought into the cache and stay in the cache after system is powered off in all cache sizes we examined in the experiments. Forced power-off attacks virtually guarantee that the system we investigate, in all configurations, will expose the secret keys in the NVM cache. This shows that a potential attacker has a reliable and reproducible mechanism to compromise disk encryption keys.

Figure 8 shows the presence of the disk encryption keys in the cache throughout the normal power-off sequence for the NoNEON and NEON systems, for different LLC sizes. We can see that the encryption key appears in the cache at roughly the

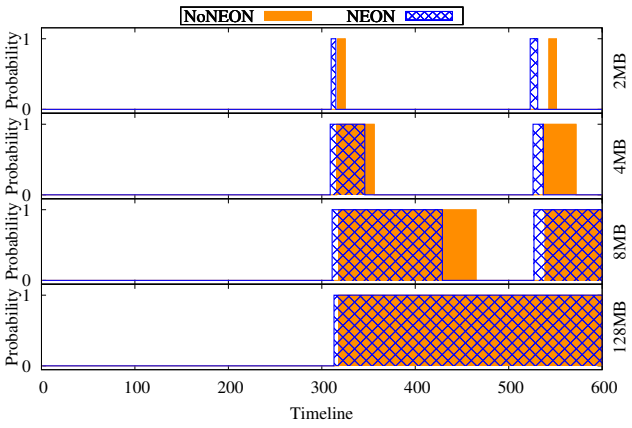

Fig. 8: AES key search outcome during normal power-off from start to completion for multiple LLC sizes.

same time following operation no. 2 in the power-off sequence (Figure 7). It is then quickly evicted in the 2MB LLC system, but persists for increasingly longer time intervals as the size of the LLC increases. For the 128MB cache, the key is never evicted before the system halts. The key is again read into the cache following operation no. 5 (unmounting the file system). In the 2MB and 4MB cases the key is again evicted before the system halts. Even for these systems an attacker could force the key to remain in the cache in a predictable way. The attacker would simply have to trigger the power-off sequence and then disconnect the processor from the power supply after a predetermined time period before the key is evicted. Since the power-off sequence is fairly deterministic, this approach has a high probability of success.

VII. Countermeasure

In order to mitigate threats of cold boot attacks against NVM caches, we propose a simple and effective software-based countermeasure. Our countermeasure is designed to force secret keys to be stored only in encrypted main memory and bypass the NVM cache throughout the execution. We develop a software API for declaring memory blocks as secrets to inform the system that their storage in the cache is not allowed. While our countermeasure applies to any secret information stored by the system, we use the disk encryption example as a case study to illustrate the concept.

A. Countermeasure Design

The process of decrypting an encrypted storage device in a system typically involves using the `cryptsetup` command-line utility in user space which calls the `dm-crypt` kernel module. This process is illustrated in Figure 9. The kernel establishes within its internal cryptographic structures the key to be used for accessing the encrypted device that has been selected via the `cryptsetup` utility. Although the process of establishing the key inside the kernel entails generating multiple copies of the key, the relevant block cipher routines in the `crypto` module dutifully use `memset()` to wipe the key after a new copy is created. As such, we only focus on the final memory location where the key is stored which is tracked by the `crypto_aes_ctx` structure. In our solution, we devise a countermeasure that is applicable to kernels that are configured to utilize hardware acceleration, as well as default kernels configured for environments where such acceleration support is unavailable.

Systems with hardware cryptographic support. Modern systems usually make use of hardware acceleration for cryptographic operations. We assume the kernel is built with the AArch64 accelerated cryptographic algorithms that make use of NEON and AES cryptographic extensions defined in the ARMv8 instruction set. This is done by including the `CONFIG_CRYPTO_AES_ARM64_*`, `CONFIG_ARM64_CRYPTO`, and `KERNEL_MODE_NEON` kernel parameters as part of the build. This translates to using architecture specific cryptographic libraries defined in `/arch/arm64/crypto` of the Linux source. In order to eliminate the presence of cryptographic keys in the cache, our solution involves marking the page associated with the address of the `crypto_aes_ctx` structure as uncacheable. We implement the necessary changes for this approach within `xts_set_key()` routine located in `aes-glue.c` where we walk through the page table in search of the appropriate page table entry that maps to the designed page. Once we locate the correct PTE, we set the `L_PTE_MT_UNCACHED` flag to label the page as uncacheable.

Systems without hardware cryptographic support. If the kernel lacks support of accelerated cryptographic hardware, we use the default cryptographic library defined in the `/crypto` directory of the Linux source. This boils down to modifying the `crypto_aes_set_key()` in `aes_generic.c`. However, we use a similar approach to the one described previously by marking the page which contains the `crypto_aes_ctx` structure to be uncacheable. The primary difference is that encryption and decryption that are used in `aes_encrypt()` and `aes_decrypt()` respectively do not make use of the 128-bit NEON registers. As such, the performance impact with this approach is higher since multiple fetches of the expanded key from memory are needed for each round of encryption or decryption.

B. Countermeasure Effectiveness

Table V summarizes the effectiveness of our countermeasure. We can see that by marking the AES key structure uncacheable our countermeasure completely eliminated the security vulnerability of NVM caches. All attack scenarios we examined are now unable to find the disk encryption keys in the cache, regardless of the benchmarks running on the system. The targeted power-off attacks also fail to identify any AES keys in the cache once the countermeasure is deployed.

C. Performance Overhead

The effectiveness of our countermeasure comes with the cost of some performance overhead. Figure 10 shows the performance overhead for different types of benchmarks executed

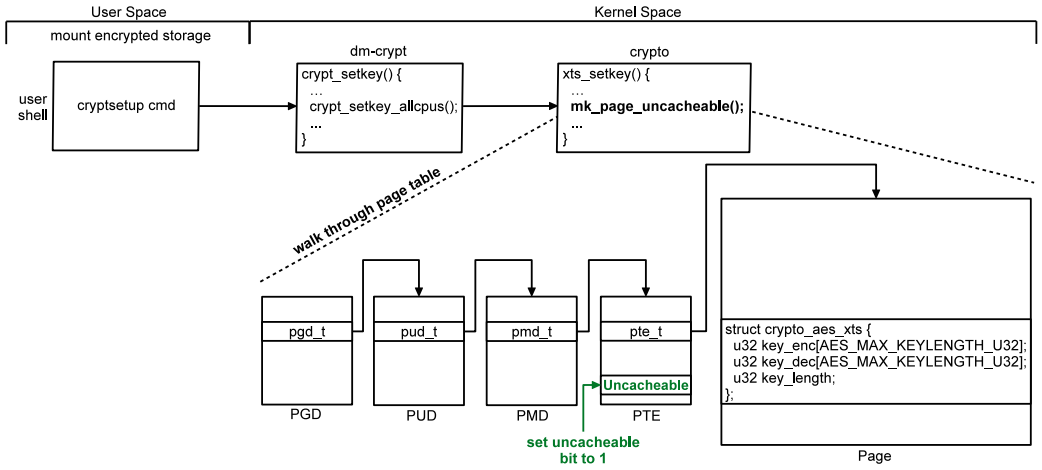

Fig. 9: Countermeasure deployment in a system with encrypted storage.

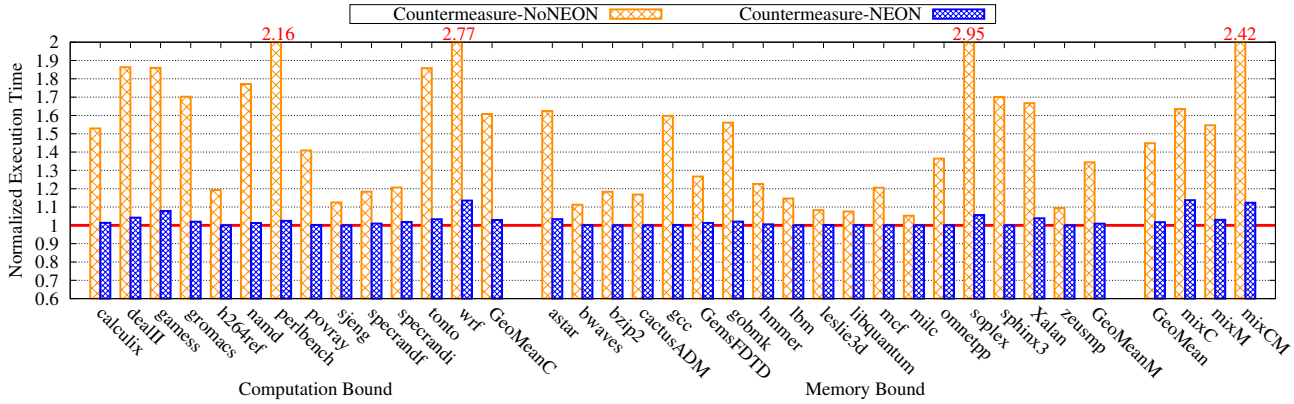

Fig. 10: Countermeasure performance overhead.

	NoNEON	NEON	Countermeasure
Single-threaded Benchmark	23 - 70%	5 - 77%	**0%**
mixC	85 - 100%	80 - 100%	**0%**
mixM	26 - 100%	20 - 100%	**0%**
mixCM	38 - 100%	34 - 100%	**0%**
Normal Power-off	0 - 100%	0 - 100%	**0%**
Forced Power-off	100%	100%	**0%**

TABLE V: Probability of finding AES keys in systems with and without the countermeasure.

in the context of our random attack. In general, the overhead for the system with NEON support is very low, averaging 2% for the single-threaded benchmarks. The overhead increases substantially if the system has no NEON acceleration – up to 45% for single-threaded benchmarks. Performance overhead of the countermeasure correlates directly with the number of encryption/decryption transactions. Since the encryption key is uncacheable, every access to the key will result in a slow memory transaction. The NEON hardware support helps alleviate this overhead substantially by storing the key in vector registers and reducing the need for memory accesses.

The performance overheads are higher as expected for multi-programmed workloads because they perform more encryption/decryption transactions overall. Overheads for the three workload mixes are 64% in NoNEON and 14% in NEON for *mixC*, 55% in NoNEON and 3% in NEON for *mixM*, and 142% in NoNEON and 12% in NEON for *mixCM*.

Performance Optimization One observation we make is that when the authenticated user is currently using the system, it is unnecessary to keep the sensitive data uncacheable since physical cold boot attacks are unlikely to be useful when the user is logged in. The attacker can extract sensitive data in more straightforward ways under those circumstances. One possible performance optimization is to enable two modes of handling secret information — cacheable and uncacheable. When user is logged in, cacheable mode on secrets is enabled so that user won't experience any performance degradation of the system. Only when the system is locked, secret information inside caches will be erased and then uncacheable mode will be turned on to protect from cold boot attacks.

VIII. Conclusion

This paper demonstrates that non-volatile caches are very vulnerable to cold boot attacks. We successfully conducted two attacks on disk encryption keys — random attacks and targeted power-off attacks. Our study shows that the probability of finding the secret AES keys in NVM caches ranges from 5% to 100% with varying workloads and cache configurations in random attacks and always reaches 100% in targeted power-off attacks. To defend computer systems against these attacks we developed a software-based countermeasure that allocates sensitive information into uncacheable memory pages. Our proposed countermeasure completely mitigates cold boot attacks against NVM caches. We hope this work will serve as a starting point for future studies on the security vulnerabilities of NVM caches and their countermeasures.

References

[1] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of Emerging Non-Volatile Memory Technologies," *Nanoscale Research Letters*, vol. 9, pp. 1–33, September 2014.

[2] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing," in *International Symposium on Computer Architecture (ISCA)*, pp. 371–382, June 2010.

[3] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *International Symposium on Computer Architecture (ISCA)*, pp. 14–23, June 2009.

[4] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the Challenges of Crossbar Resistive Memory Architectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, February 2015.

[5] X. Pan and R. Teodorescu, "NVSleep: Using Non-Volatile Memory to Enable Fast Sleep/Wakeup of Idle Cores," in *International Conference on Computer Design (ICCD)*, pp. 400–407, October 2014.

[6] X. Pan, A. Bacha, and R. Teodorescu, "Respin: Rethinking Near-Threshold Multiprocessor Design with Non-Volatile Memory," in *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 265–275, May 2017.

[7] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 239–249, February 2009.

[8] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *International Symposium on Computer Architecture (ISCA)*, pp. 2–13, June 2009.

[9] Everspin. https://www.everspin.com.

[10] Crossbar. https://www.crossbar-inc.com.

[11] Micron-Intel, "3D XPoint™ Technology." https://www.micron.com.

[12] Hewlett-Packard-Enterprise, "The Machine." https://www.hpe.com.

[13] 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," in *USENIX Security Symposium*, pp. 45–60, July 2008.

[14] M. Henson and S. Taylor, "Memory Encryption: A Survey of Existing Techniques," *ACM Computing Surveys (CSUR)*, vol. 46, April 2014.

[15] S. Chhabra and Y. Solihin, "i-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption," in *International Symposium on Computer Architecture (ISCA)*, pp. 177–188, June 2011.

[16] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 313–324, February 2017.

[17] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *International Symposium on Microarchitecture (MICRO)*, pp. 339–350, December 2003.

[18] R. Lee, P. Kwan, J. McGregor, J. Dwoskin, and Z. Wang, "Architecture for Protecting Critical Secrets in Microprocessors," in *International Symposium on Computer Architecture (ISCA)*, pp. 2–13, June 2005.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 168–177, November 2000.

[20] 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 *International Symposium on Microarchitecture (MICRO)*, pp. 183–196, December 2007.

[21] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-Efficient Encryption for Non-Volatile Memories," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 33–44, March 2015.

[22] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. D. Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting Data on Smartphones and Tablets from Memory Attacks," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 177–189, March 2015.

[23] T. Muller, A. Dewald, and F. C. Freiling, "AESSE: A Cold-Boot Resistant Implementation of AES," in *European Workshop on System Security*, pp. 42–47, April 2010.

[24] T. Muller, F. C. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *USENIX Security Symposium*, pp. 17–32, August 2011.

[25] J. Gotzfried and T. Muller, "ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices," in *International Conference on Availability, Reliability and Security (ARES)*, pp. 161–168, September 2013.

[26] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with Private Keys without RAM," in *Annual Network and Distributed System Security Symposium (NDSS)*, pp. 1–15, February 2014.

[27] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-Assisted Secure Execution on ARM Processors," in *IEEE Symposium on Security and Privacy (SP)*, pp. 72–90, May 2016.

[28] J. Gotzfried, T. Muller, S. Nurnberger, and M. Backes, "RamCrypt: Kernel-Based Address Space Encryption for User-Mode Processes," in *Asia Conference on Computer and Communications Security*, pp. 919–924, May 2016.

[29] A. Bacha and R. Teodorescu, "Authenticache: Harnessing Cache ECC for System Authentication," in *International Symposium on Microarchitecture (MICRO)*, pp. 1–12, December 2015.

[30] P. Simmons, "Security through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption," in *Annual Computer Security Applications Conference*, pp. 73–82, December 2011.

[31] J. Pabel, "Frozen Cache." http://frozencache.blogspot.com.

[32] N. A. Anagnostopoulos, S. Katzenbeisser, M. Rosenstihl, A. Schaller, S. Gabmeyer, and T. Arul, "Low-Temperature Data Remanence Attacks Against Intrinsic SRAM PUFs." Cryptology ePrint Archive, Report 2016/769, 2016.

[33] "Advanced Encryption Standard," *National Institute of Standards and Technology, Federal Information Processing Standards Publication 197*, November 2011.

[34] A. Berent, "Advanced Encryption Standard by Example." http://www.infosecwriters.com/Papers/ABerent_AESbyExample.pdf.

[35] T. Muller and M. Spreitzenbarth, "FROST: Forensic Recovery of Scrambled Telephones," in *International Conference on Applied Cryptography and Network Security*, pp. 373–388, June 2013.

[36] ARM, "DS-5 Development Studio." https://developer.arm.com.

[37] ARM, "DSTREAM High Performance Debug and Trace." https://developer.arm.com.

[38] ARM, "Juno ARM Development Platform." https://developer.arm.com.

[39] "The XTS-AES Tweakable Block Cipher," *IEEE Std 1619-2007*, May 2007.

[40] N. Binkert *et al.*, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, May 2011.

[41] J. Lell, "Practical Malleability Attack Against CBC-Encrypted LUKS Partitions." http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions.

[42] A. Jaleel, "A Pin-Based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites." http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.