
MEMSHIELD: GPU-ASSISTED SOFTWARE MEMORY ENCRYPTION

A PREPRINT

Pierpaolo Santucci*	Emiliano Ingrassia[†]	Giulio Picierro[‡]	Marco Cesati[§]
Epigenesys s.r.l., Rome, Italy		University of Rome Tor Vergata, Rome, Italy	
{santucci, ingrassia}@epigenesys.com		{giulio.picierro, cesati}@uniroma2.it	

December 16, 2019

ABSTRACT

Cryptographic algorithm implementations are vulnerable to Cold Boot attacks, which consist in exploiting the persistence of RAM cells across reboots or power down cycles to read the memory contents and recover precious sensitive data. The principal defensive weapon against Cold Boot attacks is memory encryption. In this work we propose MemShield, a memory encryption framework for user space applications that exploits a GPU to safely store the master key and perform the encryption/decryption operations. We developed a prototype that is completely transparent to existing applications and does not require changes to the OS kernel. We discuss the design, the related works, the implementation, the security analysis, and the performances of MemShield.

Keywords Data security · Memory encryption · Cryptography on GPU.

1 Introduction

Strong cryptographic algorithms rely on sensitive data that must be kept hidden and confidential. Nowadays, the most practical attacks against data encryption programs, full disk encryption layers (FDEs), authentication protocols, or secure communication channels are based on memory disclosure techniques that extract keys, hidden configurations, or unencrypted data directly from the RAM memory cells of the running systems. Consider, for an example, the (in)famous OpenSSL's Heartbleed vulnerability [12] that in 2014 allowed a remote attacker to extract secret keys of X.509 certificates used for SSL/TLS secure network protocols. Even worse, also properly implemented cryptographic programs are usually vulnerable to the class of *Cold Boot attacks* [54]. Basically, cryptographic programs rely on memory protection mechanisms implemented by the operating system kernel to hide the sensitive data from any unauthorized user or process in the system. The underlying assumptions, however, are that (1) overriding the operating system control implies rebooting the system, and (2) rebooting the system implies disrupting the contents of all RAM cells. While assumption (1) is correct if the operating system is bug-free, unfortunately assumption (2) does not generally hold. Modern RAM technologies, such as dynamic RAM (DRAM) and static RAM (SRAM), are based on electric charges stored in small capacitors that could retain the charge for several seconds, or even minutes, after power off. The typical Cold Boot attack, therefore, consists of power cycling the system using the reset button and

*ORCID iD: <https://orcid.org/0000-0003-3596-7046>.

[†]ORCID iD: <https://orcid.org/0000-0002-6710-3392>.

[‡]ORCID iD: <https://orcid.org/0000-0003-4079-8435>. Corresponding author.

[§]ORCID iD: <https://orcid.org/0000-0002-7492-3129>.

quickly rebooting from a removable device into a program that extracts the RAM contents [20]. Cold Boot attacks are so effective that they are nowadays adopted even in digital forensic activities [5, 51].

Many possible mitigations of Cold Boot attacks have been proposed in the past few years, however there still does not exist a full, practical solution. In fact, some proposals are based on cryptographic hardware circuits integrated in the system, which are still not widely adopted [29, 26, 27]. Other proposals are bound to specific hardware architectures [36, 19, 21, 55]. Moreover, many solutions are not completely transparent to the applications [53, 39, 55]. Finally, almost all proposed solutions require changes to the operating system kernel [7, 40, 16, 25, 24].

In this work we introduce *MemShield*, a framework based on a general-purpose Graphic Processing Unit (GPU) that encrypts the system memory so as to mitigate the effects of Cold Boot attacks. GPU's benefits of widespread adoption and their massive parallelism may satisfy real-time demands of transparent encryption systems, providing that communication overheads are reasonable.

MemShield is designed so as to overcome most limitations of the previous proposals. In particular, MemShield (i) relies on a vanilla Linux kernel and does not require kernel patches; (ii) it is not bound to a specific hardware architecture; (iii) it does not require specific cryptographic hardware, and uses widely adopted GPUs as secure encryption key store and cryptographic processor; (iv) it can run on legacy systems; (v) it stores encrypted data in system RAM, thus not limiting the amount of protected pages; (vi) it allows users to select the applications to be protected; (vii) it does not require source code changes in the protected applications; (viii) it exploits dynamic linking and does not require program recompilation; (ix) it achieves transparent memory accesses without code instrumentation; (x) it uses a modular software architecture for the cryptographic components, thus permitting to easily change, enhance, or upgrade the encryption cipher; (xi) it is an open-source project, available at: <https://gitlab.com/memshield/memshield/>.

We developed a prototype based on the GNU/Linux operating system and CUDA GPUs. It provides transparent memory encryption in user mode by using the userfaultfd Linux kernel framework. This prototype is a proof-of-concept aimed at testing the validity of MemShield's design, as well as assessing both security aspects and performances. As far as we know, MemShield is the first framework that achieves system memory encryption by using a GPU. Moreover, while it relies on a vanilla Linux kernel, it does not require patches to the kernel: this is also a novelty for this kind of frameworks.

The article is organized as follows: in section 2 we compare MemShield with other solutions proposed in literature. In sections 3 and 4 we describe design and implementation of MemShield. In section 5 we discuss a security analysis, while in section 6 we illustrate how MemShield impairs the overall performances of the protected applications. Finally, in section 7, we draw some conclusions.

2 Related works

Memory disclosure attacks, and in particular Cold Boot attacks [20, 2, 54], are nowadays a real menace that might expose sensible information and precious data to unauthorized use or abuse. Therefore, many researchers have proposed technical solutions to mitigate the effects of this kind of attacks.

Some proposals are based on the assumption that the attacker might be interested in a small portion of sensitive data, like the FDE master key used for encrypting a physical disk partition, or the private key of an open SSL connection. In these cases, an effective solution might be storing these data outside RAM, that is, in volatile hardware registers. For example, an AES encryption key could be stored in special CPU registers, such as x86_64 SSE registers (paranoid [35], AESSE [33]), debug registers (TRESOR [36]), profiling registers (Loop-Amnesia [42]), or ARM NEON registers (ARMORED [19]). Another approach is storing sensitive data in locked-down static-RAM caches or hardware transactional memories [46, 31, 18]. However, storing sensitive data outside system RAM does not scale up, so this solution is unfeasible when the data to be protected have size larger than the storage provided by the available volatile hardware registers.

An easy-to-implement mitigation for Cold Boot attacks aimed at large portions of sensitive data might be based on the power-on firmware of the system. For instance, during the initialization sequence the firmware might ask for a user password, or it might erase the contents of all RAM cells [45]. However, these protections can be overridden, because it is generally possible to change the firmware settings by resetting them to a default configuration, for instance by inserting a jumper on the board or removing the CMOS battery [17]. Even more crucially, the Cold Boot attack might be executed by transplanting the memory chips in a suitable system with a custom firmware [50, 51]. It has been shown [17] that cooling down the memory chips to approximately 5 Celsius degree before transplanting them allows an attacker to recover about 99% of the original bits.

Thus, even if Cold Boot attacks can be mitigated, they cannot completely ruled out: if resourceful attackers gains access to the physical RAM chips of a running system, they are able to access the system memory contents. Therefore, the most effective protection against Cold Boot attacks is based on *memory encryption*, where the contents of some (or, possibly, all) memory cells are encrypted by using one or more random keys stored in volatile hardware registers outside of the RAM chips. In order to get meaningful information from the memory dump, the attacker must either recover the encryption key(s) from the volatile hardware registers, or break the encryption cipher. Henson et al. [22] present a survey of hardware enhancements to support memory encryption at memory controller, bus, and caches level, protecting data transfers against eavesdropping.

Ideally, memory encryption could be implemented at hardware level so as to be completely transparent to user mode applications and, in some cases, even to the operating system kernel. Nowadays, major chip manufacturers design processors with MMU-based memory encryption. For instance, Intel Software Guard Extensions (SGX) [27] allows user applications to create an encrypted memory area, called *enclave*, for both code and data. Encrypted pages are stored in a page cache of limited size. Confidentiality and integrity are guaranteed even when the OS kernel is compromised. However, SGX is not designed for full memory encryption and legacy applications cannot use it without source code changes. In 2017 Intel announced Total Memory Encryption (TME) [26], a hardware extension to encrypt all RAM cells with either a global key or multiple keys; in any case, the encryption key cannot be retrieved by software. TME is currently under development. AMD’s proposed solution is named Secure Memory Encryption (SME) [29]: it encrypts and decrypts pages of memory if a specific flag in the page table entries is set. The encryption key is randomly generated by the hardware and loaded at boot time into the memory controller. An extension called TSME allows the hardware to perform memory encryption transparently to the OS and the user software. Mofrad et al. [34] analyze features, security attacks, and performances of both Intel SGX and AMD SME solutions.

While there is a raising interest to add support for MMU-based memory encryption solutions in the operating system kernels [41], these systems are still not widespread, because this technology is rather expensive and operating system support is scarce. Thus, in the past years several researchers have proposed to encrypt the memory by using frameworks exploiting either common-of-the-shelf (COTS) hardware components or custom devices. Because the MMUs lack circuits aimed at encryption, all these solutions must rely on procedures included in hypervisors or operating system kernels, so they can be collectively named *software-assisted memory encryption* frameworks. In this work we propose MemShield, an innovative software solution based on COTS hardware.

In software-assisted memory encryption, a typical access to a protected page triggers a chain of events like the following: (i) the ordinary MMU circuits generate a hardware event (e.g., a missing page fault); (ii) the user application that made the access is suspended and an OS kernel/hypervisor handler schedules the execution of a cryptographic procedure (either on a CPU, or on some custom device like a FPGA or a GPU); (iii) the cryptographic procedure encrypts or decrypts the data in the accessed page; (iv) the OS kernel/hypervisor resumes the execution of the user application. A crucial point of this mechanism is that the encryption keys used by cryptographic ciphers must not be stored “in clear” on the system memory, otherwise an attacker might retrieve them and decrypt all RAM contents. Hence, software-assisted memory encryption proposals generally include, as a component, one of the already mentioned solutions for storing a limited amount of sensitive data outside of system RAM [36, 42, 35, 46, 31, 18].

We can reasonably expect that the performances of software-assisted memory encryption are significantly worse than those of MMU-based memory encryption. However, software-assisted memory encryption has potentially several advantages over MMU-based memory encryption: (i) it can be used on legacy systems, as well as in low-level and mid-level modern systems without dedicated circuits in the MMU; (ii) it might be less expensive, because CPU/MMU circuits without cryptographic support are smaller and simpler; (iii) it is much easier to scrutiny its design and implementation in order to look for vulnerabilities and trapdoors; (iv) it is much easier to fix vulnerabilities and trapdoors, or to enhance the cryptographic procedures if the need arises.

One of the first proposal for software-assisted memory encryption, although motivated by avoiding bus snooping rather than Cold Boot attacks, is in Chen et al. [7]: this framework uses locked-down static cache areas or special scratchpad memories (usually found in embedded hardware) as a reserved area for encrypted data. This solution requires changes in the operating system kernel to support memory access detection and to avoid that data in the static cache are leaked into system RAM. The authors, however, do not discuss how to protect the memory encryption key from disclosure. A similar idea was explored in [53]: the authors describe a FPGA prototype named Exzess that implements a PCIe board acting as a transparent memory proxy aimed at encryption. The encrypted data are stored on the Exzess device itself: device memory is mapped on the address space of a process, so that read and write accesses on those pages trigger decryption and encryption operations on the device, respectively. A drawback of this approach is that the size of the “encrypted RAM” is limited in practice by the capacity of the FPGA board. MemShield is quite different than these solutions because it stores the encrypted data on the system RAM.

CryptKeeper [40] is a closed-source extension to the Linux kernel swapping mechanism: user pages can be flagged as “encrypted” and, when their number in RAM raises above a given threshold, removed from RAM and stored in encrypted form in a swap area. However, CryptKeeper is fragile versus Cold Boot attacks because it stores the encryption key in RAM. A related idea is in Huber et al. [25]: the authors suggest to perform encryption of user space processes memory at suspend time, using the same key used for Full Disk Encryption (FDE). Yet another variant of this idea is presented in [24], where the system memory of portable devices, like notebooks and smartphones, is encrypted by means of the “freezer” infrastructure of the Linux kernel. MemShield is aimed at protecting against Cold Boot attacks possibly performed when the system is up and running, so it has a rather different design.

Henson and Taylor [21] describe a solution based on ARM architecture that implements a microkernel exploiting a cryptographic microprocessor to handle encrypted data either in a scratchpad memory or in the system RAM. An improvement of this idea is in [55], where the ARM TrustZone ensures that unencrypted data never leak to system RAM. The authors use the ARM Trusted Execution Environment to execute a microkernel that loads encrypted program from RAM, decrypt them, and run them safely. The framework requires patches to the general-purpose OS kernel, Linux, that handles the non-trusted programs.

RamCrypt [16] is an open-source Linux kernel extension aimed at encrypting memory at page-level granularity. It is transparent to the user applications because the decryption is automatically triggered by the page fault handler whenever an user access is attempted. RamCrypt protects anonymous pages and non-shared mapped pages of the applications. In order to ensure acceptable performances, a small number of the last recently accessed pages is kept in clear in a “sliding window” data structure. RamCrypt also takes care to avoid key leaks in RAM by using a slightly modified AES implementation from TRESOR [36]. Even if TRESOR has been shown to be vulnerable to some classes of attacks [4], RamCrypt is still an effective mitigation against Cold Boot attacks. MemShield adopts some ideas from RamCrypt, mainly the encryption at page level triggered by page faults and the sliding window mechanism; however, MemShield does not require patches to operating system kernel, and it makes use of a GPU as a safe store and processor for the cryptographic operations.

HyperCrypt [15] is similar to RamCrypt, however memory encryption is handled by a hypervisor (BitVisor) rather than a kernel program; this allows HyperCrypt to also protect kernel pages. Like RamCrypt, HyperCrypt is based on TRESOR [36]. TransCrypt [23] is similar to HyperCrypt, yet it relies on ARM Virtualization Extensions to implement a tiny encryption hypervisor.

Papadopoulos et al. [39] proposed a framework for software-assisted memory encryption that can protect either part of, or the whole system memory; the master key is kept in CPU debug registers, like in [36]. The framework relies on code instrumentation to intercept load/store instructions that access memory, and it requires some patches to the operating system kernel.

Finally, EncExec [8] makes use of static caches as storage units for encryption keys and unencrypted data. Whenever a user application accesses an encrypted page, EncExec decrypts the data and locks them in the static cache so that the application can transparently get the unencrypted data. EncExec adopts an interesting approach, however it requires patches to the operating system kernel.

Concerning the usage of the GPU for safely executing cryptographic routines, the seminal work is PixelVault [48], which implements AES and RSA algorithms on the GPU by carefully storing the cryptographic keys inside the GPU registers. The main goal of that work is to implement a more secure version of the OpenSSL library in order to mitigate memory disclosure attacks. PixelVault is based on a GPU kernel that runs forever waiting for crypto operation requests submitted via a memory regions shared between CPU and GPU. As discussed in [56], PixelVault is actually vulnerable to unprivileged attackers, however several authors suggested ways to enhance its approach. For instance, the authors in [52] propose to run CUDA cryptographic applications inside guest VMs by using a virtualized GPU; no privileged attacker on guest VMs is able to retrieve the encryption keys, because they are never stored in the guest VM memory. In [49] the authors suggest to modify the GPU hardware to prevent the device driver from directly accessing GPU critical internal resources. In [28] the authors propose to use a custom interface between GPU and CPU and to extend the Intel SGX technology to execute the GPU device driver in a trusted environment that a privileged attacker cannot access. MemShield mitigates Cold Boot attacks, so it assumes that privileged users are trusted (consider that a privileged user might easily get a full memory dump without a Cold Boot attack). Therefore, MemShield just ensures that unprivileged users cannot access the GPU, thus avoiding the original PixelVault vulnerabilities.

3 Design overview

MemShield is designed to mitigate Cold Boot attacks perpetrated by malicious actors with physical access to the machine. We assume trusted privileged users and processes, as well as safe operating system and base programs.

The rationale behind MemShield’s design was to provide memory encryption services to multiple concurrent users logged on the same machine, using a COTS GPU as secure key store and cryptographic processor. Thus, MemShield does not rely on custom cryptographic circuits or specific hardware architectures.

GPUs are massively parallel accelerators consisting in thousand of cores typically grouped in several *compute units*. Cores in the same compute unit can communicate via *shared memory*, typically implemented as a fast user programmable cache memory. Different compute units can communicate each other via *global memory*, that is, the GPU’s RAM. In GPU programming terminology, CPU and system RAM are referred as *host*, while GPU and its RAM are referred as *device*. Device code is written in special functions called *kernels*. Those functions, once invoked from the host, can trigger multiple parallel executions of the same kernel function over the input, depending on how the kernel is launched.

MemShield transparently encrypts user-space memory at page granularity and decrypts them on-demand whenever an access is attempted. There is no need to change or rebuild the source code of the protected applications. Moreover, users may select the programs to be protected, so that applications that do not handle sensitive data do not suffer any slowdown.

By design, for each process and at any time, MemShield enforces a bounded number of pages in clear, while keeping most of them encrypted. This mechanism is based on a *sliding window* and it is an effective solution against Cold Boot attacks, as already proved in [16]. Encrypted data are stored in system RAM, hence there is virtually no limit on the amount of pages that can be protected.

MemShield’s core is a daemon that is in charge of encrypting and decrypting memory on behalf of clients. We define as *client* any process interested in memory encryption services. Because GPU programming is supported by user mode tools and libraries, MemShield daemon runs in user mode.

To support transparent memory encryption, the client that attempts to access an encrypted page must be suspended and decrypted contents for that page must be provided. In order to achieve transparent memory encryption, MemShield must be able to detect clients’ attempts to access encrypted memory and provide decrypted contents. As a matter of fact, detection of memory accesses represents one of the most challenging aspects of the project. We chose to address this issue by using *userfaultfd* [47], a framework recently added to the Linux kernel aimed at efficient page fault handling in user space. Thanks to *userfaultfd*, no changes are required to the operating system kernel. Currently *userfaultfd* is used by applications that implement memory checkpoint/restore functionality [14] and live migration of both containers [43] and virtual machines [44]. MemShield is the first project that uses *userfaultfd* for memory encryption, as far as we know.

In systems with a Memory Management Unit (MMU), the *logical addresses* appearing in CPU instructions must be translated in *physical addresses* to be used for RAM’s chips programming. Translations are described by means of the *page tables*, usually a per-process data structure kept by the OS kernel. Each page table entry contains a page’s virtual-to-physical mapping and attributes such as protection bits (Read, Write, eXecute). A *page fault* is generated by the MMU hardware if a translation is not found (*missing fault*) or the wrong access type is attempted (*protection fault*). This mechanism is actually used by operating systems to isolate process address spaces. In all POSIX-compliant systems there exists an established technique to detect memory accesses in user space: it consists of changing the page permissions so as to forbid any access, then executing a custom handler for the SIGSEGV signal sent by the kernel to the process at any access attempt [13, 6]. However, this mechanism may significantly impair the performances of the clients compared to *userfaultfd*.

A crucial design goal of MemShield is transparency with respect to its clients: no change to the source code of the clients is required, and no recompilation is needed. In order to achieve this goal, we assume that clients are dynamically linked to the C library (*libc*); in practice, this is almost always true. The idea is to intercept calls to the memory management functions typically provided by the *libc* implementation, in order to register the client’s memory areas to *userfaultfd*. The custom handlers of the memory management functions are collected in a user-mode library called *libMemShield*. This library can be loaded in the client’s process before the C library using the LD_PRELOAD technique, so that the custom handlers override the original library symbols.

Because MemShield must handle several concurrent clients with a single GPU, *userfaultfd* works in non-cooperative mode: a single user-mode process is notified of any page fault events occurring in registered clients. We call this process the MemShield’s *server*.

The server does not perform cryptographic operations directly, because it cannot access the encryption keys stored in the GPU registers. The most straightforward way to implement this mechanism is to launch on the GPU an always-running kernel that implements a safe cipher.

3.1 Design limitations

By design, MemShield handles only private anonymous memory: it is not concerned with memory areas backed by files or shared among different processes. It is also based on `userfaultfd`, which has some constraints of its own: mainly, it cannot protect the memory area handled by `brk()` or `sbrk()`. However, MemShield can handle the client’s stack, any memory area obtained by `malloc()`, `calloc()`, `realloc()`, as well as the anonymous memory obtained by `mmap()` and `remap()`: typically, sensitive data end up being stored in such pages.

4 Implementation details

The main activities performed by MemShield are: (i) memory area registration to `userfaultfd`, (ii) page fault handling, (iii) sliding window management, and (iv) GPU cryptographic operations.

4.1 Memory area registration

When a client allocates a memory area to be protected, MemShield must register the corresponding set of virtual addresses to `userfaultfd`. In order to achieve this, `libMemShield` overrides some C library functions.

For allocations performed by anonymous memory mapping (`mmap()` and analog functions), the custom wrapper just performs the original procedure and registers the obtained virtual address to `userfaultfd`.

On the other hand, handling memory areas obtained by `malloc()` and similar functions is more demanding, because the C library might use the `brk()/sbrk()` system calls to perform the allocation. MemShield forces the C library to always use anonymous memory mapping when allocating memory by means of the `mallopt()` tuning function.

The `userfaultfd` framework does not handle stack pages. To overcome this problem, `libMemShield` replaces original stack pages with memory allocated with `mmap()` using `sigaltstack()`. Since stack encryption could have a significant impact on overall performances of MemShield, encrypting stack pages can be selectively enabled or disabled by means of an environment variable.

`libMemShield` also overrides the `free()` and `munmap()` functions, because it ensures that any page is filled with zeros before releasing it to the system. Of course, this is crucial to avoid leaking sensitive data in RAM.

4.2 Page fault handling

At the core of MemShield there is the virtual address space mechanism implemented by the operating system. Any client is allowed to access a given number of anonymous pages “in clear”, that is, in unencrypted form. These pages belong to the address space of the client and the corresponding physical page frames are referenced in the page tables. On the other hand, the page table entries associated with encrypted pages denote missing physical pages. The physical pages storing the encrypted data belong to the server’s address space, and are referenced in a per-client red-black tree sorted by virtual addresses.

When a client accesses an encrypted page, the MMU raises a page fault because of the missing physical page in the client’s page table. Since the corresponding virtual address has been registered to `userfaultfd`, the server is notified about the event. As a consequence, the server looks for the virtual address of the missing page in the client’s red-black tree and retrieves the physical page containing the encrypted data. Then, the server sends the encrypted data to the GPU, which performs the decryption operation. Subsequently, the server relies on `userfaultfd` to resolve the client’s page fault by providing a physical page containing the decrypted data.

If the server does not find a virtual address in a client’s red-black tree, the page is missing because it has never been accessed earlier. Therefore, the server allocates a new red-black tree node and resolves the client’s page fault by providing a physical page containing all zeros.

4.3 Sliding window management

The server keeps a per-client data structure called sliding window, which is a list of virtual addresses corresponding to unencrypted anonymous pages of the client. The sliding window maximum size is configurable. When the server is going to resolve a page fault, it adds the corresponding virtual address to the sliding window. It also checks whether its size has become greater than a preconfigured maximum. In this case, it takes away the oldest unencrypted page in the sliding window, which is thus removed from the client’s page tables.

The server cannot operate on the address space of the client. Therefore, libMemShield creates at initialization time a thread called *data thread*, which acts on the client’s address space on behalf of the server. Correspondingly, a thread is created in the server to handle the requests of the client concurrently with those of the other clients.

Server and data thread exchange information by means of a Unix socket and a shared memory area. The Unix socket is used only to transmit open file descriptors and control messages. MemShield does not send sensitive data with this socket, because the Linux implementation makes use of a kernel memory buffer which could be vulnerable to Cold Boot attacks. The shared memory area, instead, is safe, because it is composed by a single page frame in RAM, which is explicitly filled with zeros by the server as soon as a transfer is completed.

When the server must drop a page from a sliding window, it sends the corresponding virtual address to the data thread. The latter provides the contents of the page, that is, the unencrypted data, to the server; then, it clears the physical pages and invokes the `MADV_DONTNEED` command of the `madvise()` system call to remove the physical page from the page tables. The server encrypts the data by means of the GPU, and adds the encrypted page to the client’s red-black tree.

4.4 GPU encryption

MemShield cryptographic module services the requests coming from the server using the GPU both as a secure key store and a secure processor. Any cryptographic procedure operates on a single page of 4096 bytes.

The data are encrypted by using the cipher ChaCha [3]. This choice was motivated by the need for a cipher that is, at the same time, (i) secure, (ii) suitable for GPU computation, and (iii) simple enough so that the computation can be performed completely in GPU registers. MemShield implements the strongest variant ChaCha20, which can be regarded as cryptographically safe [32, 10, 9].

The actual encryption/decryption of a 4096-byte page is performed by XORing the data with several keystream blocks. ChaCha20 computes a 512-bit block of keystream starting from a 384-bit seed composed by a 256-bit key and a 128-bit value. In MemShield the 256-bit key is unique and it is generated by a cryptographically secure pseudo-random number generator provided by the operating system. This key is sent to the GPU and stored only in GPU registers, afterwards it is purged out of the server memory.

The 128-bit value of the seed, which can be used both as a counter in stream ciphering and as a predefined nonce, is composed by the virtual address of the page, the process identifier (PID) of the client, and a counter ranging from 0 to 63 that corresponds to the index of the 512-bit block inside the page. Observe that the keystream blocks could be generated and XORed independently with the plaintext blocks. The ciphertext construction is thus embarrassingly parallel, which is a highly desirable feature in a GPU implementation. Another useful property is that encryption and decryption are performed with the same operations, hence the GPU kernel can use the same function for both.

MemShield cryptographic module makes use of a NVIDIA GPU programmed by means of the CUDA toolchain [37, 38]. The GPU is reserved to MemShield, which means that unprivileged users cannot access the device. In practice, because the communication channel between user space and the CUDA driver is based on device files, the permissions of these device files are changed so that access is only allowed to privileged users.

The GPU kernel consists of several CUDA blocks; each block acts as a *worker* whose job is to extract pages from a queue and process them using 32 CUDA threads (one *warp*). Each CUDA thread generates two ChaCha20 keystream blocks (128 bytes), which are then XORed with the same amount of plaintext. The number of CUDA blocks in the GPU kernel is dynamically computed at run time according to the features of the GPU board. Using more than one block allows the server to submit requests for several concurrent clients.

Because the 256-bit encryption key is created at initialization time and stored inside the GPU registers, the GPU kernel cannot be terminated, otherwise the key would be lost. MemShield uses a mapped memory between host and device to implement a shared data structure that controls the GPU operations. The data transfer between host (server) and device (GPU) is realized through a circular buffer implementing a multiple-producer, single-consumer queue: multiple host threads can submit concurrent pages to the same queue, while those will be processed by a single worker on the GPU. Each worker has its own queue, thus the workers runs independently and concurrently.

An important aspect of the implementation is that the encryption key and the internal state of the cipher are never stored in GPU local memory, otherwise MemShield would be vulnerable to GPU memory disclosure attacks. We verified that the current implementation of the GPU kernel never does register spilling.

4.5 Prototype limitations

The current implementation of MemShield is a prototype, thus it has some limitations. First of all, any protected application must be single process. There is no major obstacle to enhance MemShield so as to overcome this limit.

Protected applications must also be single thread. It would be possible to extend MemShield to support multi-threaded processes whenever `userfaultfd` becomes capable of handling write-protect page faults. Work is in progress to integrate this feature in the vanilla Linux kernel [1].

MemShield protects all private anonymous pages. In order to have better performances, it could be preferable to selectively encrypt only the subset of pages containing sensitive data.

Finally, the ChaCha20 implementation is prototypal and could be improved.

5 Security Analysis

MemShield is an effective mitigation against Cold Boot attacks. In fact, RamCrypt’s authors already proved [16] that the sliding window mechanism is an effective technique that could drastically reduce the probability to find meaningful encryption keys or other sensitive data in memory dumps. Like RamCrypt, MemShield makes use of one sliding window per each protected application.

MemShield is also inspired by how PixelVault [48] makes use of the GPU to safely store encryption keys and run cryptographic procedures. However, PixelVault is nowadays assumed to be vulnerable [56, 11]. The reported attack vectors to PixelVault were based on launching malicious kernel functions on the GPU, or running a CUDA debugger on the running GPU kernels. MemShield avoids these vulnerabilities because it restricts access to the GPU to privileged users. Recall that, in our threat model, privileged users and privileged processes are always regarded as trusted. MemShield also takes care of avoiding GPU register spilling, so that Cold Boot attacks against the GPU memory would not retrieve any sensitive data of the cryptographic procedure.

Memory dumps obtained by Cold Boot attacks might expose data included in the kernel buffers associated to Unix sockets, pipes, or other process communication channels. Actually, we verified that the Linux implementation of Unix sockets is vulnerable, because the kernel never erases the associated buffers. MemShield carefully avoids sending sensitive data by means of Unix sockets. Rather, it makes use of shared memory, whose contents can be explicitly cleared by MemShield at the end of the sensitive data transmission.

MemShield does not weaken the operating system isolation guarantees at runtime, thus the confidentiality of users’ data against malicious users logged on the same system is preserved. In particular, observe that a unprivileged user cannot tamper with the server daemon, because we assumed that the operating system is safe and the server is a privileged process.

A malicious user could try to interfere with MemShield’s communication channels. The daemon listens on a Unix socket waiting for connection requests from clients; hence, unprivileged processes must have write permissions on this socket. However, an attacker cannot replace the socket (in order to mount a man-in-the-middle attack), because the socket interface file is placed in a directory owned by root and not writable by unprivileged users.

A malicious user cannot even tamper with the shared memory area used to exchange data between client and daemon: in fact, this area is created by means of file descriptors exchanged privately through the Unix socket; the area is then mapped in the address spaces of both daemon and client.

MemShield is not designed to protect sensitive data against DMA attacks or other side channel attacks; just consider that a successful DMA attack might break the whole operating system guarantees, while in our threat model the operating system is assumed to be sound.

6 Performance Evaluation

In order to establish the performance impact of MemShield prototype on protected applications, we ran some benchmarks. All tests have been executed on a workstation equipped with a 3.5 GHz Intel Core i7 4771 CPU having 4 physical and 8 logical cores, 32 GiB RAM, and a GPU NVIDIA GeForce GTX 970 (compute capability 5.2) with 4 GiB of device memory. The workstation used Slackware 14.2 as Linux distribution with kernel 4.14, *glibc* version 2.23, NVIDIA driver version 418.67 and CUDA 10.1. CPU power-saving was disabled.

Two benchmarks (*crypt*, and *qsort*) belong to the stress-ng [30] test suite, version 0.09.48. *crypt* consists of multiple invocations of the C library `crypt_r()` function on data placed onto the stack. The test is executed by:


```
$ stress-ng --crypt 1 --crypt-ops 1000
```

qsort performs multiple sorts of an array of elements allocated with `calloc()`. To sort the array, the C library function `qsort()` is used:

```
$ stress-ng --qsort 1 --qsort-size 2048 --qsort-opts 10000
```

The third benchmark is *aes* from *OpenSSL* suite version 1.0.2s, which operates on data structures stored in pages allocated with several calls to `malloc()` interleaved with calls to `free()`. The test consists of encrypting the Linux kernel 5.0 source archive, placed in a RAM-based filesystem, using AES-256 in CBC mode:

```
$ openssl enc -aes-256-cbc -in linux-5.0.tar.gz -out /dev/null -k pass
```

Finally, the fourth benchmark is *sha512* from *GNU Coreutils* 1.25: it invokes `malloc()` to allocate a single buffer storing file data, then it computes the digest by storing cryptographic internal state on the stack. The test was launched on the Linux kernel 5.0 source archive placed in a RAM-based filesystem:

```
$ sha512sum linux-5.0.tar.gz
```

In every test run we collected the execution times by using GNU *time* 1.7; each specific test has been repeated 10 times, then average values and standard deviations have been computed.

Table 1 reports how MemShield affects the average execution times of the four benchmarks with different sliding window configurations. Figure 1 shows the slowdowns of the four benchmarks with respect to the baseline, which is the running time without MemShield protection. In order to better understand how the different components of MemShield contribute to the overhead, any plot has four lines, which correspond to the following cases: (i) encryption on GPU of all private anonymous pages, including the stack (“GPU, Stack”), (ii) encryption on GPU of all private anonymous pages, excluding the stack (“GPU, no Stack”), (iii) no encryption at all (the GPU is not involved in handling the protected pages, thus MemShield server stores the pages in clear), but handling of all private anonymous pages, including the stack (“no GPU, Stack”), and finally (iv) no encryption at all, for all private anonymous pages, excluding the stack (“no GPU, no Stack”). Distinguishing between “Stack” and “no Stack” slowdowns is important because, when an application does not have sensitive data stored on the stack, disabling stack encryption significantly improves the performances of MemShield. Distinguishing between “GPU” and “no GPU” slowdowns is useful in order to understand how much the userfaultfd-based mechanism impairs, by itself, the performances of the protected applications. Note that, even if the GPU kernel is a component that can be easily replaced, for instance by an implementation of another, more efficient cipher, transferring the data of the protected pages between system RAM and GPU has an intrinsic cost that could not be easily reduced in the current MemShield implementation.

The *aes* benchmark has a very large slowdown for sliding window size less than 16. A typical AES implementation makes use of very large tables, which are continuously and randomly accessed. The plot shows that the accesses fall both into many `malloc()`ed pages and, to a lesser extent, into the stack. Reducing the sliding window size from eight to four significantly increases the slowdown, which means that the accesses to the encrypted pages are quite random. GPU encryption roughly doubles the slowdown values.

The *sha512* benchmark has negligible slowdown for sliding window sizes greater than eight. The plot shows that the overhead is due mainly to anonymous private pages not included in the stack, that is, stack handling does not contribute a lot to the slowdown. In fact, the program uses several `malloc()`ed pages as a buffer for data read from file, while it uses the stack to store a small data structure for the digest inner state. Decreasing the sliding window size from eight to four does not significantly change the slowdown, which means that the replacement policy of the sliding window is working fine: actually, the program tends to read sequentially the pages in the buffer. GPU encryption doubles the slowdown values.

		aes	sha512	qsort	crypt
Baseline		0.30 ± 0.03	0.44 ± 0.01	4.67 ± 0.03	4.76 ± 0.02
Sliding window	32	1.76 ± 0.02	0.47 ± 0.01	6.07 ± 0.02	12.53 ± 0.08
	16	1.94 ± 0.01	0.48 ± 0.00	6.08 ± 0.04	12.56 ± 0.09
	8	16.64 ± 0.09	4.39 ± 0.02	6.10 ± 0.03	12.56 ± 0.10
	4	35.23 ± 0.22	4.58 ± 0.03	64.35 ± 1.62	12.52 ± 0.08

Table 1: Wall-clock execution times of single-instance benchmarks, with different sliding window sizes and without MemShield protection (“Baseline”). Average times and standard deviations are in seconds. Sliding window sizes are in pages. MemShield encrypts all private anonymous pages, including the stack.

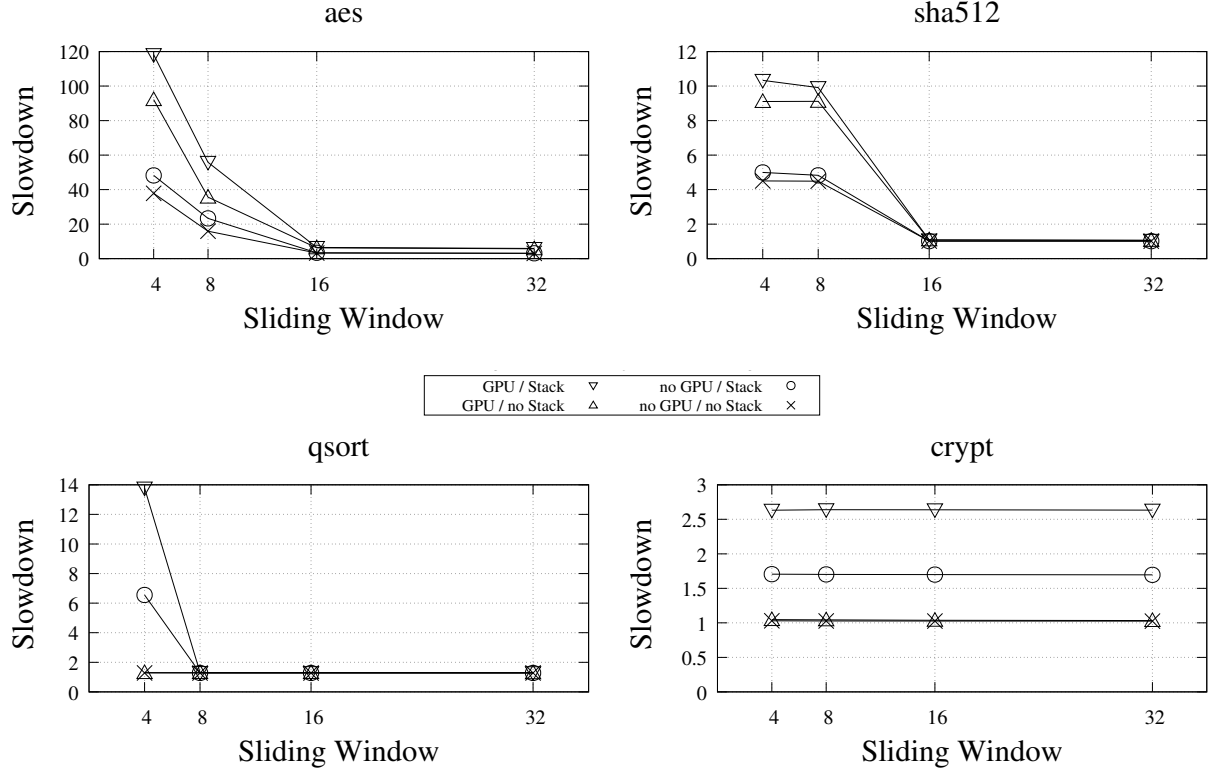


Figure 1: Slowdowns of a single instance of the benchmarks relative to baseline (no MemShield) varying the sliding window size. MemShield protects all private anonymous pages including the stack (“Stack”) or excluding it (“no Stack”). Encryption is performed by the GPU (“GPU”) or not done at all (“no GPU”).

The *qsort* benchmark has been selected so to emphasize the effects of stack encryption in some types of protected applications. The program sorts 2048 integers eventually using the C library function `qsort()`. This function avoids recursion by storing on the stack some pointers to the array partitions still to be sorted. Using a sliding window of four pages, page fault handling causes a slowdown roughly seven when the stack is protected, and the GPU encryption doubles this value. On the other hand, if stack is not included in the protected pages, the slowdown is always negligible.

		aes	
		Baseline	MemShield
Inst.	1	0.30 ± 0.03	35.23 ± 0.22
	2	0.30 ± 0.01	36.77 ± 0.16
	4	0.31 ± 0.01	39.91 ± 0.47

		sha512	
		Baseline	MemShield
Inst.	1	0.44 ± 0.01	4.58 ± 0.03
	2	0.46 ± 0.01	4.82 ± 0.03
	4	0.48 ± 0.01	5.28 ± 0.06

		qsort	
		Baseline	MemShield
Inst.	1	4.67 ± 0.03	64.35 ± 1.62
	2	4.74 ± 0.03	67.26 ± 1.23
	4	4.94 ± 0.01	73.37 ± 1.26

		crypt	
		Baseline	MemShield
Inst.	1	4.76 ± 0.02	12.52 ± 0.08
	2	4.81 ± 0.04	13.15 ± 0.14
	4	5.03 ± 0.02	15.04 ± 0.10

Table 2: Wall-clock execution times of concurrent instances of the benchmarks, without MemShield (“Baseline”) and with MemShield using a sliding window of four pages. Average times and standard deviations are in seconds. “Inst.” denotes the number of concurrent benchmark instances. MemShield encrypts all private anonymous pages, including the stack.

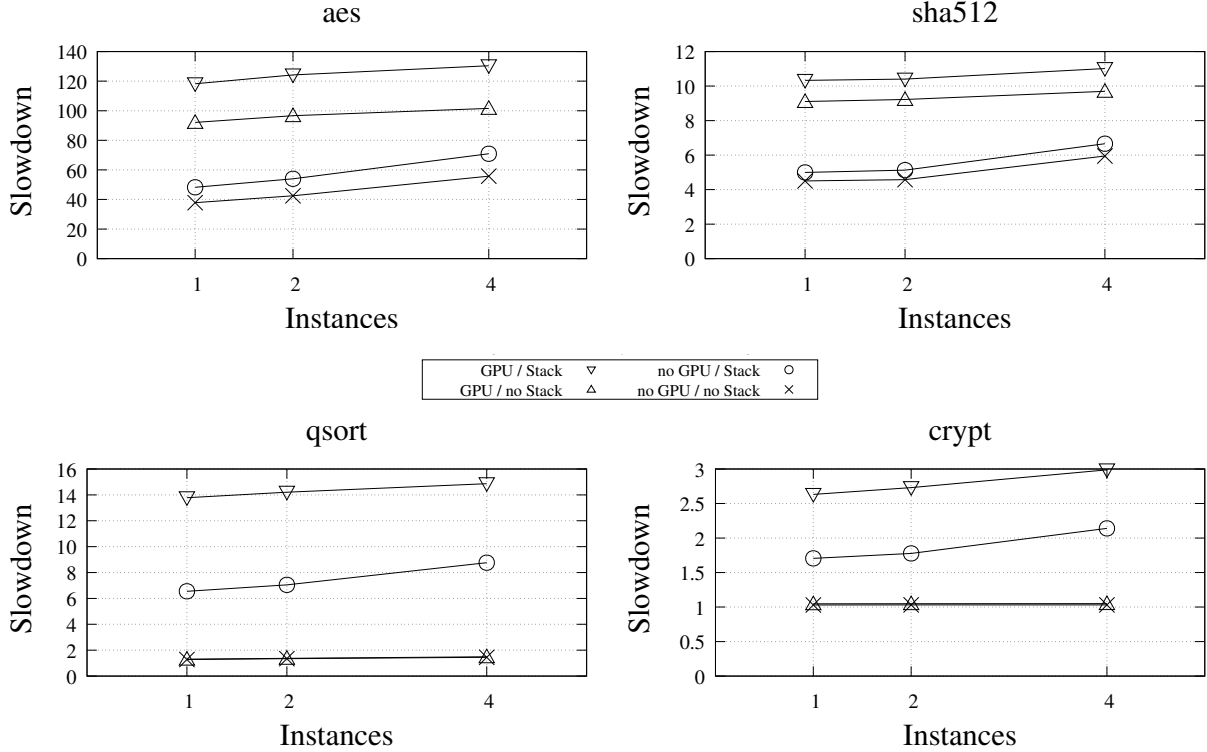


Figure 2: Slowdowns relative to baseline (no MemShield) varying the number of benchmark instances. MemShield protects all private anonymous pages including the stack (“Stack”) or excluding it (“no Stack”), with a sliding window of 4 pages. Encryption is performed by the GPU (“GPU”) or not done at all (“no GPU”).

Similarly, the *crypt* benchmark has a significant slowdown only when the stack is encrypted. In fact, the *crypt_r()* function is invoked on data placed on the stack. The slowdown caused by GPU encryption, by itself, is roughly equal to the slowdown caused by page fault handling. The overhead of MemShield is quite small, for every sliding window size.

We also run another set of benchmarks to verify how MemShield scales up when the number of protected clients increases. Each benchmark launches one, two, or four instances of the program at the same time; observe that our test machine has only four physical cores. As shown in Table 2, average execution times have been collected with and without MemShield protection, with sliding window size equal to four pages and stack encryption enabled. Figure 2 shows the slowdowns of the average execution times with respect to the baseline, that is, execution without MemShield protection. According to the plots, there is a limited increase of the slowdown when the number of concurrent instances grows, both for page fault handling and for GPU encryption.

7 Conclusions

Memory encryption is an effective solution against memory disclosure attacks, in which an attacker could access and dump system RAM contents, in order to extract any sort of sensitive data. In this article we presented MemShield, which is a novel approach to software memory encryption for user-mode applications that uses a GPU as a secure key store and crypto processor. By ensuring that the key and the internal state of the cipher are stored into GPU hardware registers, MemShield guarantees that this sensitive information are never leaked to system RAM; therefore, the attacker cannot get meaningful data from a system dump unless the encryption cipher is broken.

Compared to all other proposals for memory encryption frameworks, MemShield is implemented by a user-mode daemon and does not require patches to the operating system kernel. Moreover, user applications do not require source code changes, recompilation, or code instrumentation, hence MemShield can protect even applications for which only the executable code is available.

Functional tests and security analysis suggest that MemShield is an effective mitigation of Cold Boot attacks aimed at system RAM. Performance measures on a prototype implementation show how the MemShield overhead heavily depends on the chosen configuration and clients' memory access patterns. Moreover, the current implementation can be significantly improved, for instance by implementing selective page encryption, by optimizing the GPU kernel implementation, or by introducing some mechanisms that start encrypting pages in the sliding window "in background" as the number of free slots becomes lower than a predefined threshold.

Acknowledgments

We gratefully thank Emiliano Betti for his valuable suggestions, support, and encouragements. The material presented in this paper is based upon work partially supported by Epigenesys s.r.l.. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of Epigenesys s.r.l..

References

- [1] Arcangeli, A.: aa.git repository. <https://git.kernel.org/pub/scm/linux/kernel/git/andrea/aa.git/>, accessed: 17 Sept 2019
- [2] Bauer, J., Gruhn, M., C. Freiling, F.: Lest we forget: Cold-boot attacks on scrambled DDR3 memory. *Digital Investigation* **16**, S65–S74 (03 2016)
- [3] Bernstein, D.J.: ChaCha, a variant of Salsa20. In: *Workshop Record of SASC*. vol. 8, pp. 3–5 (2008)
- [4] Blass, E.O., Robertson, W.: TRESOR-HUNT: Attacking CPU-bound encryption. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. pp. 71–78. ACSAC '12, ACM, New York, NY, USA (2012)
- [5] Carbone, R., Bean, C., Salois, M.: An in-depth analysis of the Cold Boot attack: Can it be used for sound forensic memory acquisition? Tech. Rep. DRDC Valcartier TM 2010-296, Defence R&D Canada - Valcartier (Jan 2011)
- [6] Cesati, M., Mancuso, R., Betti, E., Caccamo, M.: A memory access detection methodology for accurate workload characterization. In: *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. pp. 141–148 (08 2015)
- [7] Chen, X., Dick, R.P., Choudhary, A.: Operating system controlled processor-memory bus encryption. In: *2008 Design, Automation and Test in Europe*. pp. 1154–1159 (March 2008)
- [8] Chen, Y., Khandaker, M.R., Wang, Z.: Secure in-cache execution. In: *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017*. pp. 381–402 (2017)
- [9] Choudhuri, A.R., Maitra, S.: Differential cryptanalysis of Salsa and ChaCha—an evaluation with a hybrid model. *IACR Cryptology ePrint Archive* **2016**, 377 (2016)
- [10] Dey, S., Sarkar, S.: Improved analysis for reduced round Salsa and Chacha. *Discrete Applied Mathematics* **227**, 58–69 (2017)
- [11] Di Pietro, R., Lombardi, F., Villani, A.: CUDA leaks: A detailed hack for CUDA and a (partial) fix. *ACM Trans. Embed. Comput. Syst.* **15**(1), 15:1–15:25 (Jan 2016)
- [12] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., Halderman, J.A.: The matter of Heartbleed. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. pp. 475–488. IMC '14, ACM, New York, NY, USA (2014)
- [13] Edelson, D.: Fault interpretation: fine-grain monitoring of page accesses. Tech. rep., University of California at Santa Cruz (1992)
- [14] Emelyanov, P.: CRIU: Checkpoint/restore in userspace (July 2011), <https://criu.org>
- [15] Götzfried, J., Dörr, N., Palutke, R., Müller, T.: HyperCrypt: Hypervisor-based encryption of kernel and user space. In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. pp. 79–87 (Aug 2016)
- [16] Götzfried, J., Müller, T., Drescher, G., Nürnberger, S., Backes, M.: RamCrypt: Kernel-based address space encryption for user-mode processes. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. pp. 919–924. ASIA CCS '16, ACM, New York, NY, USA (2016)
- [17] Gruhn, M.: Forensically sound data acquisition in the age of anti-forensic innocence (Nov 2016), Ph.D. Thesis, Der Technischen Fakultät der Friedrich-Alexander-Universität Erlangen-Nürnberg

- [18] Guan, L., Cao, C., Zhu, S., Lin, J., Liu, P., Xia, Y., Luo, B.: Protecting mobile devices from physical memory attacks with targeted encryption. In: Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks. pp. 34–44. WiSec’19, ACM (2019)
- [19] Gtzfried, J., Miller, T.: ARMORED: CPU-bound encryption for Android-driven ARM devices. In: 2013 International Conference on Availability, Reliability and Security. pp. 161–168 (Sept 2013)
- [20] Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM* **52**(5), 91–98 (May 2009)
- [21] Henson, M., Taylor, S.: Beyond full disk encryption: protection on security-enhanced commodity processors. In: International Conference on Applied Cryptography and Network Security. pp. 307–321. Springer (2013)
- [22] Henson, M., Taylor, S.: Memory encryption: A survey of existing techniques. *ACM Comput. Surv.* **46**(4), 53:1–53:26 (Mar 2014)
- [23] Horsch, J., Huber, M., Wessel, S.: TransCrypt: Transparent main memory encryption using a minimal ARM hypervisor. In: 2017 IEEE Trustcom/BigDataSE/ICSS. pp. 152–161 (Aug 2017)
- [24] Huber, M., Horsch, J., Ali, J., Wessel, S.: Freeze and Crypt: Linux kernel support for main memory encryption. *Computers & Security* **86**, 420–436 (2019)
- [25] Huber, M., Horsch, J., Wessel, S.: Protecting suspended devices from memory attacks. In: Proceedings of the 10th European Workshop on Systems Security. pp. 10:1–10:6. EuroSec’17, ACM, New York, NY, USA (2017)
- [26] Intel®: Memory encryption technologies specification. Tech. rep., Intel Corp. (April 2019)
- [27] Intel®: Software Guard Extensions, accessed: 9 Sept 2019
- [28] Jang, I., Tang, A., Kim, T., Sethumadhavan, S., Huh, J.: Heterogeneous isolated execution for commodity GPUs. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 455–468. ASPLOS ’19, ACM (2019)
- [29] Kaplan, D., Powell, J., Woller, T.: AMD memory encryption whitepaper (2016)
- [30] King, C.: stress-ng test suite (2011), <https://kernel.ubuntu.com/~cking/stress-ng>
- [31] Lin, J., Guan, L., Ma, Z., Luo, B., Xia, L., Jing, J.: Copker: A cryptographic engine against cold-boot attacks. *IEEE Transactions on Dependable and Secure Computing* **PP** (11 2016)
- [32] Maitra, S.: Chosen IV cryptanalysis on reduced round ChaCha and Salsa. *Discrete Applied Mathematics* **208**, 88–97 (2016)
- [33] Müller, T., Dewald, A., Freiling, F.: AESSE: a cold-boot resistant implementation of AES. In: Proc. Third European Workshop on System Security, EUROSEC’10. pp. 42–47 (2010)
- [34] Mofrad, S., Zhang, F., Lu, S., Shi, W.: A comparison study of Intel SGX and AMD memory encryption technology. In: Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 9:1–9:8. HASP ’18, ACM, New York, NY, USA (2018)
- [35] Müller, T.: Cold-Boot resistant implementation of AES in the Linux kernel (May 2010), master thesis, RWTH Aachen University
- [36] Müller, T., Freiling, F.C., Dewald, A.: TRESOR runs encryption securely outside RAM. In: USENIX Security Symposium. vol. 17 (2011)
- [37] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. *Queue* **6**(2), 40–53 (Mar 2008)
- [38] NVIDIA®: CUDA toolkit, <https://developer.nvidia.com/cuda-toolkit>
- [39] Papadopoulos, P., Vasiliadis, G., Christou, G., Markatos, E., Ioannidis, S.: No sugar but all the taste! Memory encryption without architectural support. In: 22nd European Symposium on Research in Computer Security, ESORICS 2017. pp. 362–380 (2017)
- [40] Peterson, P.A.H.: CryptKeeper: Improving security with encrypted RAM. In: 2010 IEEE International Conference on Technologies for Homeland Security (HST). pp. 120–126 (Nov 2010)
- [41] Rybczyska, M.: A proposed API for full-memory encryption (January 2019), <https://lwn.net/Articles/776688>
- [42] Simmons, P.: Security through Amnesia: A software-based solution to the Cold Boot attack on disk encryption. *Computing Research Repository - CORR* (04 2011)
- [43] Stoyanov, R., Kollingbaum, M.J.: Efficient live migration of Linux containers. In: International Conference on High Performance Computing. pp. 184–193. Springer (2018)

- [44] Suetake, M., Kizu, H., Kourai, K.: Split migration of large memory virtual machines. In: Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems. pp. 4:1–4:8. APSys '16, ACM, New York, NY, USA (2016)
- [45] TCG platform reset attack mitigation specification. Tech. rep., Trusted Computing Group (2008), <https://www.trustedcomputinggroup.org/wp-content/uploads/Platform-Reset-Attack-Mitigation-Specification.pdf>
- [46] Tews, E.: Frozencache—mitigating cold-boot attacks for full-disk-encryption software (Dec 2010), 27th Chaos Communication Congress
- [47] Userfaultfd. Man page on kernel.org: <https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt>, accessed: 30 Aug 2019
- [48] Vasiliadis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Pixelvault: Using GPUs for securing cryptographic operations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1131–1142. ACM (2014)
- [49] Volos, S., Vaswani, K., Bruno, R.: Graviton: Trusted execution environments on gpus. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 681–696. USENIX Association, Carlsbad, CA (Oct 2018)
- [50] Vömel, S., C. Freiling, F.: A survey of main memory acquisition and analysis techniques for the Windows operating system. *Digital Investigation* **8**, 3–22 (07 2011)
- [51] Vömel, S., C. Freiling, F.: Correctness, atomicity, and integrity: Defining criteria for forensically-sound memory acquisition. *Digital Investigation* **9**, 125137 (11 2012)
- [52] Wang, Z., Zheng, F., Lin, J., Dong, J.: Utilizing GPU Virtualization to Protect the Private Keys of GPU Cryptographic Computation, pp. 142–157 (10 2018)
- [53] Würstlein, A., Gernoth, M., Götzfried, J., Müller, T.: Exzess: Hardware-based RAM encryption against physical memory disclosure. In: Hannig, F., Cardoso, J.M.P., Pionteck, T., Fey, D., Schröder-Preikschat, W., Teich, J. (eds.) *Architecture of Computing Systems – ARCS 2016*. pp. 60–71. Springer International Publishing, Cham (2016)
- [54] Yitbarek, S.F., Aga, M.T., Das, R., Austin, T.: Cold Boot attacks are still hot: Security analysis of memory scramblers in modern processors. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 313–324 (Feb 2017)
- [55] Zhang, M., Zhang, Q., Zhao, S., Shi, Z., Guan, Y.: Softme: A software-based memory protection approach for tee system to resist physical attacks. *Security and Communication Networks* **2019**, 1–12 (03 2019)
- [56] Zhu, Z., Kim, S., Rozhanski, Y., Hu, Y., Witchel, E., Silberstein, M.: Understanding the security of discrete GPUs. In: Proceedings of the General Purpose GPUs. pp. 1–11. GPGPU-10, ACM, New York, NY, USA (2017)