Protecting Secrets of Persistent Systems with Volatility

Vasily A. Sartakov
IBR TU Braunschweig
sartakov@ibr.cs.tu-bs.de

Rüdiger Kapitza
IBR TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

***Abstract*—The volatility of main memory and CPU caches is an important implicit protection mechanism for sensitive data: in-memory data gets erased if memory modules are disconnected from power supply. Persistent systems, on the other hand, cannot rely on volatility and without further measures their secrets can be easily retrieved by physical access.**

In this paper, we present Volatility, a system which protects secrets stored in persistent memory. This system provides mechanisms which turn persistent sub-systems into volatile ones by the use of AMD Secure Memory Encryption (SME), a new extension of AMD CPUs which provides encryption of main memory at the page granularity. Volatility protects secrets at two levels: it offers fine-grained memory encryption inside the kernel, where only information considered as sensitive is secured, and per-process memory encryption, which encrypts selected user space programs. Besides storing subsystems in an encrypted form, all relevant input and output paths, e.g. managed by the kernel, are protected as well. Our evaluation of Volatility demonstrates that the proposed protection mechanism does not impact the system performance, while protecting against strong adversaries.

***Index Terms*—Memory encryption, Persistent systems, NV-RAM, AMD SME, Volatile-by-encryption**

I. Introduction

Non-Volatile RAM (NV-RAM) is considered as a replacement for conventional Dynamic RAM (DRAM) [1]. Indeed, prospective persistent memory technologies such as Spin Transfer Torque Magnetoresistive RAM (STT-MRAM) [2] show a latency comparable to DRAM, while offering a larger capacity [3]. Moreover, NV-RAM has an important property, *persistency*, since this memory can retain stored data in the absence of an external source of power. Together these features allow to consider future computing systems as *fully* persistent systems, i.e. equipped by NV-RAM only.

Fully persistent systems have advantages over hybrid systems since the latter need to combine two different memory technologies, while the former, in turn, would result in simpler hardware and system support. However, in the case of fully persistent systems, the volatility of main memory is lost, and in this paper, we argue that this feature of memory is crucial and needs to be preserved in persistent systems. To support this claim, let us consider the content of memory from the security point of view.

Main memory contains various secrets, such are storage encryptions keys, cached decrypted data, and more. DRAM protects this information from the direct extraction: the volatile nature of memory destroys the data as soon as an attacker removes DRAM modules from a memory slot. The volatility can be overcome, but direct extraction of data requires the freezing of the equipment [4], which sometimes is hard to perform. Persistent memory, in turn, does not require low temperatures to retain this data and the content of memory modules can be rather easily extracted.

In this paper, we present Volatility, a software system aimed at the protection of secrets in persistent systems from physical attacks. To ensure this, firstly, we introduce a mechanism for reliable erasing of persistent data. This mechanism uses the SME extension of recent AMD processors and enables creation of Volatile-by-Encryption (VbE) memory regions. Such regions are encrypted transparently by an inaccessible, volatile encryption key, which is stored inside the memory controller and regenerated on each boot of the system.

Secondly, we demonstrate, that system components dealing with security-sensitive data can produce side objects, which also need to be protected since they may contain parts of the data. As a consequence, Volatility additionally protects all side artefacts, as well as all relevant data paths in the system. For that, we developed a mechanism for tracking interactions of system components with security-sensitive data and turned the identified system components into Volatile-by-Encryption.

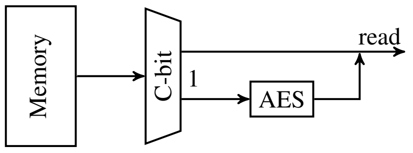
Thirdly, we implemented and evaluated a complex use case on top of Volatility. This use case requires various system components for data processing, in particular, data input/output, block layer, caches, and encryption subsystem. In sum, the evaluation demonstrated almost negligible performance overhead, while preserving the persistence of the system and the protection of secrets.

The paper has the following structure. Section II provides background on memory encryption, persistent systems, and demonstrates how security sensitive data is stored in multiple places of the kernel. Section III overviews the design of Volatility. Section IV discusses the implementation of Volatility. Section V and Section VI outline a methodology and system support for identification of security data paths in the kernel. Section VII presents our evaluation of Volatility. Finally, Section VIII overviews related works while Section IX concludes the paper.

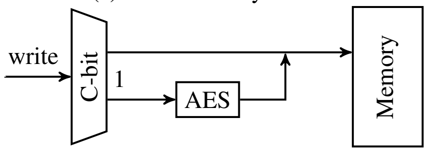
II. Background

A. Data path of sensitive information

The core concept of Volatility is that it protects not only security-sensitive user space programs, but all relevant data paths inside the kernel. These paths may include copies of data used by the programs. To support this claim, let us consider as an example the *login* utility.



(a) Read memory routine



(b) Write memory routine

Fig. 1: AMD Secure Memory Encryption

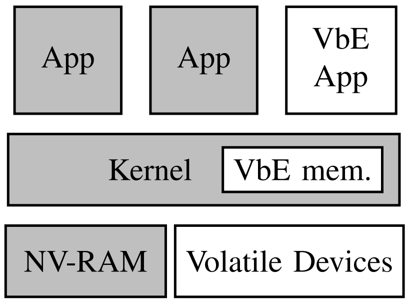


Fig. 2: Simplified architecture of Volatility

The *login* program deals with security sensitive information, such as the *password credentials*, and therefore needs to be protected. The utility reads a password from input devices such as a keyboard and stores it inside a dynamically allocated memory region. This memory region can be encrypted on per-page basis. However, only encrypting this region is not sufficient: To demonstrate this, we created a memory image of a commodity Linux instance immediately after the successful login and found two places where the plaintext password was still located in memory. One of these places belonged to the *login* utility itself, while another one resided in the Linux kernel memory. Thus, it is not enough to only secure application memory to successfully prevent the data retrieval, but *some* components of the kernel should be encrypted as well. In the scope of this paper, we refer to these additional components as *Security Sensitive Data Set (SSDS)*. This is described in detail in Section III-C.

B. Secure Memory Encryption

Volatility is based on hardware-accelerated memory encryption, such as provided by Secure Memory Encryption (SME). The latter is currently surfacing on the market as part of AMD's Ryzen processors [5]. This technology enables encryption and decryption of main memory content on a per-page basis. The cryptographic operations are performed and implemented as part of the memory controller. As an encryption algorithm AES is used and encryption keys are generate inside the CPU each time the system restarts. These keys cannot be read or written by software.

a) Accessing encrypted and plain memory.: Figure 1a and Figure 1b show the general schemes of how encrypted memory is accessed. The memory controller has two options to access physical memory: as part of the first one the memory controller directly communicates with the memory modules to perform read and write operations. The second one involves using the AES engine for de- and encryption of memory pages. The memory controller chooses the access method in accordance with a special bit (called C-bit – *enCrypted*) in the Page Table Entry (PTE) of an accessed virtual address.

b) SME operation modes.: SME supports two modes of operation: Transparent SME (TSME) and page-wise selective memory encryption. While in the first case the whole main memory is transparently encrypted, the second case has been designed to selectively encrypt main memory. Especially the first mode of operation can be used instantaneously to protect secrets, but at the same time, the whole system will not be persistent anymore.

C. Persistent Systems

The memory hierarchy of "classical" systems includes three key layers: CPU caches, primary storage and secondary storage. CPU caches are the fastest, but small, while primary storage is slower, but more capacious, and secondary storage is the slowest, but the most capacious. The first and second layers are volatile, i.e. do not retain stored information without an external source of power, whereas the third layer is non-volatile. Also, primary storage is byte-addressable, while secondary storage is block-addressable. New memory technologies, such as Phase-Change RAM (PC-RAM) [1] and Spin Transfer Torque Magnetoresistive RAM (STT-MRAM) [2], are at the same time capacious, non-volatile and byte-addressable. Therefore, instead of three layers, persistent systems likely have only two layers in the future: volatile CPU caches and non-volatile main memory. Currently, there are several different conceptions of persistent systems so-called *models of persistence*. A model of persistence describes systems components which should be persistent, the interaction of volatile devices with persistent components, life-cycles of programs and more. One can identify four models of persistence. Firstly, there is a language- or library-based model of persistence, introduced by Mnemosyne [6] and NV-Heaps [7]. In this model, a persistent system is considered as a hybrid one, where both types of memory are presented. The majority of such systems parts are volatile, while components of user programs can be allocated in persistent memory.

Another model of persistence was introduced by NV-Process [8]. In this model, the entire process can be allocated inside persistent memory. The *System-wide* model generalised [9] this approach to the whole system. In this approach, there is only persistent memory, and all components of it are persistent per se. Finally, there is the *hypervisor-based* model of persistence [10], [11], which provides transparent persistence at the level of virtual machines.

III. The design of Volatility

The main goal of Volatility is to offer protection of sensitive information, while also maintaining the persistence. For that, Volatility provides a set of operating system primitives including *fine-grained in-kernel* and *coarse-grained per-process memory encryption* of persistent systems.

In this section, we detail the core concepts of Volatility. Firstly, we describe hardware assumptions and a model of persistence. Secondly, we explain the assumed attacker model. Thirdly, we describe encryption-based volatile processes. Finally, we detail Security Sensitive Data Sets, our approach to protect sensitive information processed outside a secured application, e.g. in the kernel space.

A. Persistence model and Volatile processes

We assume, that the hardware platform includes only NV-RAM (Figure 2). Therefore, the software system follows the *system-wide* model of persistence and all processes and the kernel are persistent per se. The hardware platform should include a power-outage detector [12], which is used for early detection of incoming power outages. This detector initialises a *flush-on-fail* routine [9], during which the system software flushes volatile caches.

The *Volatile-by-Encryption (VbE) processes* are encrypted persistent processes. Such processes differ from ordinary persistent processes, as they cannot be recovered after a power outage. Moreover, all dynamic allocations that are made in the context of a VbE process are also volatile, i.e. automatically encrypted. This concerns not only dynamically allocated objects in user space but also includes objects created by the kernel in the context of the target process. These VbE objects are typically caches, structures, memory buffers and more, allocated by the kernel when execution system calls on behalf of the protected process. VbE objects, as well as VbE processes, are not recoverable, and each encrypted subsystem has its own recovery hook, which cleanups VbE memory upon restart.

B. Attacker model

Volatility aims to prevent direct extraction of sensitive information from NV-RAM. It relies on the availability of page-wise hardware memory encryption, as offered by SME [5]. Based on these initial considerations, we define the following attacker model:

Firstly, the target system offers page-wise hardware memory encryption, which is free of flaws. The system software manages encryption of security sensitive information. The encryption key for physical memory encryption is securely created inside the CPU at startup time and cannot be retrieved from CPU.

Secondly, we assume that the target system uses a Volatility-enabled operating system. The system software consists of security sensitive and security insensitive data. Security sensitive programs are protected by page-wise hardware memory encryption. Security insensitive programs are not protected by memory encryption and can be recovered by an intruder.

Thirdly, the intruder does not have access to login credentials and cannot obtain root privileges by exploiting operating system vulnerabilities, using a malicious device or performing a side-channel attack. All these cases should be treated as different kind of attacks and they require additional protection measures to be handled. Indeed, if the attacker can obtain root privileges, they can extract data without physical access.

C. Security Sensitive Data Set

The encryption of process memory is typically not enough to provide full-fledged protection of sensitive data. For example, input and output paths via kernel objects should be encrypted as well. In this paper we call these additional paths as SSDSs.

The SSDS is a set of system software (libraries, programs, kernel modules or components), whose encryption is required to protect sensitive data of an application. In the previously considered example, to protect the *login* utility one needs to encrypt all buffers inside the kernel which store input data.

Different applications (as well as kernel components) have different SSDSs. For example, for storage encryption, one needs to protect file caches and the encryption context inside the kernel. While for a Graphical User Interface (GUI) application one needs to protect input/output buffers inside the kernel and intermediate buffers inside the window manager. In user space, applications' security sensitive data can be protected easily with coarse-grained encryption, but SSDSs inside the kernel require additional modifications.

We analysed several possible usage patterns of per-process encryptions and identified two groups of Linux kernel subsystems belonging to different SSDSs. We call the first group of components *Default SSDS*, while we refer to the second group of components as *Demand SSDS*.

Components from the *Default* group are always encrypted. These components are lightweight and shared by multiple programs or kernel subsystems. Serial input is an example of such a lightweight subsystem shared by multiple programs.

Components belonging to the *Demand* group are also encrypted, but the encryption is performed only for one particular application. For example, many programs create file caches, but only the caches belonging to encrypted storages should be encrypted. In this group, we also add subsystems that are encrypted by default, but not shared by multiple subsystems. For example *dm-crypt*, which offers transparent disc encryption that is only used by the *device mapper*. Demand and Default SSDSs are described in Sections V-A to V-C.

IV. Implementation

This section first outlines how support for SME-usage has been added to the Linux kernel. Next, it is detailed how a VbE process dynamically marks other objects for encryption. We conclude the section with a description of a testing framework to validate that sensitive data of a protected application is in fact secured.

A. SME support in the Kernel

Before detailing the actual implementation of Volatility's kernel-level support for SME, we shortly analyse the Linux kernel memory hierarchy: The lowest level of the Linux kernel memory hierarchy is represented by architecture specific memory-related structures. On this level for example one can change bit fields inside the architecture specific Page Table Entry (PTE) or flush the Translation Lookaside Buffer (TLB) cache. These bit fields define whether a memory page is executable or non-executable, writable or read-only and, in our case, encrypted or plaintext.

The middle level of the memory hierarchy is associated with the function *alloc_page()*. This function allocates one frame of physical memory and maps it to a vacant virtual address, after which it returns a *struct page* – an object describing the performed mapping.

The *alloc_page()* function requests one argument – a bit field which defines the way how the memory will be allocated, for example, whether the memory allocator should use cold or warm pages. Conceptually, this function does not allow specifying the kind of allocated frame, like non-executable or read-only. This is performed by calling architecture-specific functions from the lowest level of the memory hierarchy.

The highest level of the memory hierarchy can be associated with the function *kmalloc()*. This function is abled to allocate objects of any size, i.e. smaller or larger than a memory frame size. This is enabled by an intermediate allocator, running on top of the page allocator.

We considered two different approaches for establishing encrypted memory support: The first approach is based on manually changing attributes of objects previously allocated via *alloc_page()* This approach targets the middle level of the memory allocation hierarchy as an implementation point. As mentioned above, each allocated page can be encrypted by modifying the corresponding PTE. The second approach is based on the idea, that encrypted memory should be allocated from a pool of encrypted pages. This approach assumes the highest level of the memory hierarchy as an implementation point.

Both approaches have their individual advantages and disadvantages. The first approach is flexible and does not require significant modification to the memory subsystem. Memory encryption can be applied to any allocated page or memory region on demand. Additionally, the encryption flag of the pages can be changed dynamically without any restrictions. However, this approach only enables the encryption of objects of page-aligned size, but do not interfere with other non-encrypted allocations.

The second approach requires the introduction of a new memory pool, which will be dedicated to encrypted memory pages. From this pool, an allocator can take objects of any size (non-page-aligned), but memory assigned to this pool will be no longer accessible for ordinary allocations. Also, it will be impossible to change attributes of objects from this poll without a threat of neighbouring objects being damaged.

Finally, we decided on the first approach and implemented memory encryption support on top of the existing page allocation support. We added a new flag, `__GFP_VOLATILE`, indicating that an allocated page needs to be encrypted after the allocation. The highest level of memory hierarchy can also be extended to support the allocation of encrypted pages, since it passes allocation flags to the middle level. To prevent corruption of neighbouring objects we round up to the full page size when an allocation is performed in conjunction with the `__GFP_VOLATILE` flag.

B. Volatile process

To indicate the volatility of a process to the Volatility extended system we use a *volatility marker*. The marker has several properties. Firstly, marked and non-marked processes must be distinguishable in kernel space, but marking itself should be performed in user space. Secondly, the marker should be inheritable, i.e. all entities created by executing *exec(ve)* and *fork* system calls should also be marked and encrypted. Thirdly, the marker should be protected by security policies of the operating system – i.e., only privileged users can mark or unmark processes/programs.

For that, we used the "POSIX capabilities in Linux" [13], which extend the existing access control infrastructure. In short, we introduced a new capability and modified the process of program loading. By default, all processes do not have the volatility capability flag set, but if the loader identifies the flag inside the extended attributes of a file, it raises the corresponding flag inside the *task_struct* and from then this process, and all subprocesses created by this "marked" process, will be encrypted.

As mentioned before, coarse-grained encryption enables encryption of all objects created by a marked process. In the Linux kernel, if the process makes a system call, the kernel performs this call in the context of the user process. Within this system call, the kernel can create new kernel objects, and they will be created by the kernel, but in the context of the process. The volatility capability can be checked [1] easily in different subsystems of the Linux kernel. Accordingly, for contexts with the volatility capability set to true, the kernel should make objects encrypted, and non-encrypted for contexts without the volatility marker. For example, system calls like *mmap*, *mprotect*, *brk*, etc. now check the capability and always allocate encrypted memory regions for volatile processes. On the low level it is achieved by the introduction of new memory protection flags, `PROT_VOLATILE` and `VM_VOLATILE`, linked with the C-bit in the PTE structure.

V. Identifying SSDS

An operating system can create additional objects, which can contain copies of security sensitive data belonging to a Volatility-protected process. As we discussed previously, the operating system can create caches or use chains of buffers for I/O operations. To provide protection to these additional

[1] if(cap_raised(cur->cred->cap_effective, CAP_VOLATILE)) {<...>};

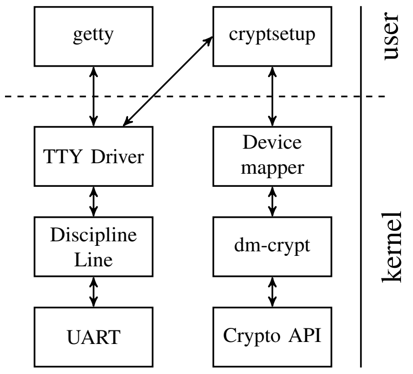


Fig. 3: Input, dm-crypt and cryptsetup

objects, we introduced Security Sensitive Data Set (SSDS) – a set of kernel-space and user space components that need to be encrypted to secure all sensitive information of an application.

This section describes modifications of three essential Linux kernel subsystems: input, dm-crypt and page cache.

A. Input

The input subsystem represents a *Default* component of SSDS. This component of the kernel is lightweight and shared by multiple programs. Furthermore, there is another reason why this subsystem should be encrypted by default: some components of input perform memory allocations in the context of an interrupt. In the interrupt context, we cannot identify whether the allocation is made for a volatile or non-volatile process. Thus, to prevent leakage of input data, we made the input subsystem a Default component of SSDS. In other words, all objects created inside this subsystem are encrypted.

As in Section III, we consider the login utility but this time from the implementation point of view. In fact, login depends on the *getty* utility, which opens a connection to a virtual console, reads login name and then calls login, which checks the provided credentials. Figure 3 shows components of the input subsystem involved into interaction with the utility. As one can see, there are four components involved in input processing.

Firstly, the 8250 UART driver[2] represent the lowest level of the input call hierarchy. This driver receives interrupts from the device and delivers key scan code to the *discipline line*. The second component, discipline line, is a special buffer where actual editing of the input is performed each time a user presses a key like backspace [14]. The third component is a *tty driver*, which provides a terminal interface to user space program. Finally, the user space utility *getty* reads the login name from the TTY device and then executes the *login* utility.

We made the following changes to the input system to enable encryption. Firstly, we marked the *getty* utility as VbE. Secondly, we encrypted the read and echo buffers of discipline line (n_tty.c) and tty_buffer inside the TTY driver. Finally, we enabled encryption for the transmit buffer *xmit* (serial_core.c) allocation, which is performed originally in the context of an interrupt.

[2]We used virtual environment and executed target system inside QEMU. That is why we used UART device as an input. More about the testing environment is described in Section VII

B. Encrypted storage

The *dm-crypt* module represents a *Default* component of SSDS. This component of the kernel is not lightweight, but implements security sensitive functionality which needs to be encrypted.

dm-crypt enables block device encryption and is implemented as a component of the Device Mapper. The Device Mapper is a kernel subsystem which enables the creation of virtual block devices such as *dm-crypt*

The *dm-crypt* does not perform cryptographic operations by itself, but uses another subsystem of the kernel, *Crypto API*. Also, the interaction with encrypted devices requires a preparation phase, when a user enters a passphrase and creates an encrypted virtual device, which can be mounted to a virtual file system. This utility executes in user space and is called *cryptsetup*. To summarise, to prevent leakage of security sensitive information, one needs to encrypt:

- input subsystem, because *cryptsetup* reads pass-phrases from the input
- components of *Crypto API*, because they are involved in the key management
- encryption contexts of *dm-crypt*, because they contain non-encrypted data and keys.

Figure 3 shows components involved in the configuration of dm-crypt. Since the input subsystem is protected as a Default SSDS, in this subsection we focus on the Device Mapper and the encryption engine.

Firstly, the *cryptsetup* utility reads a passphrase from a console. Then, *cryptsetup* configures the device mapper by creating a new entry inside a mapping table and specifies the kind of the mapping engine as *dm-crypt*. Thirdly, *dm-crypt* communicates with the Crypto API, recovers the encrypted master key from the storage and decrypts it with the passphrase. After that, *dm-crypt* can perform I/O encryption or decryption on a per-sector basis. For such operations, *dm-crypt* prepares a structure for crypto requests which includes memory regions for encrypted and decrypted forms of a sector, a master key, pointers on encryption context *crypto_skcipher* and additional data. Then, the module calls the *Crypto API*.

To protect *dm-crypt*, we perform the following actions. Firstly, we mark *cryptsetup* as VbE. Secondly, we manually encrypt memory for encryption requests: *crypt_config* and *skcipher_request* structures and transform mask structure *crypto_tfm*[3].

C. Page cache

The Linux kernel uses different caches to increase performance. For example, the kernel uses a SLAB allocator to allocate new objects of the same-size from an object pool, a page cache for caching block device accesses and others.

[3]Allocated inside __crypt_alloc_tfm()

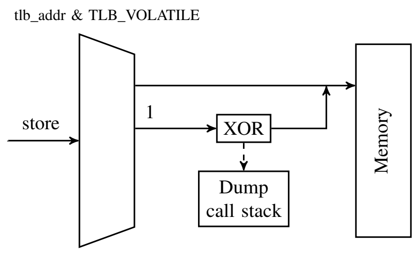


Fig. 4: Store routine in QEMU

Some caches should be part of SSDS since they could contain security sensitive data. At the same time, encryption of all caches leads to performance degradation and data corruption. Thus, we need to selectively encrypt caches where necessary.

In our use case, we propose to use encrypted storage and thus, we are interested in the protection of the associated *page cache*. *Page cache* is a cache of storage pages in RAM: each time when the kernel needs to perform read/write operations it checks whether the data is located in the page cache. If so, the data is fetched from there, otherwise, the kernel populates the cache and then operates with cached data [15].

In contrast to *input* and *dm-crypt*, the *page cache* represents a *Demand* component of SSDS: the *Page cache* is not lightweight, and cached objects do not always contain security sensitive data.

As described previously, we use a *volatility marker* to distinct VbE processes from the normal processes. This marker can be tracked easily inside any component of the Linux kernel, in particular, inside the page cache. In accordance with this marker, any subsystem can change its own behaviour, for example, modify memory allocation flags or use additional subroutines.

In this way, we implemented cache encryption of VbE processes. Each time the page cache allocates memory for a new cache, it checks whether the `CAP_VOLATILE` flag is raised or not. If raised, then the subsystem performs memory allocation with the `__GFP_VOLATILE` flag. In sum, we implemented encryption for two kinds of caches: *readahead*[4] and *writeback*[5].

D. Other subsystems

We considered and modified three subsystems: input, dm-crypt and page cache. However, more complex applications which involve GUI or networking will require analysis and modifications other subsystems. This can be performed similarly.

VI. Basic SME support in QEMU

For development purposes, we added basic SME support to the QEMU virtualisation engine. This was necessary to verify the accuracy of SSDS tracking and due to the absence of persistent SME-enabled platform on the market.

In particular, we added new routines for page-level encryption and decryption. These routines handle the C-bit of the Page Table Entry, when virtual addresses are accessed. To do so, we modified the TLB emulation and extended the support for load/store operations. Moreover, we implemented a tracking mechanism which identified all kernel functions that access encrypted memory.

A. Encrypted memory support

To enable SME emulation in QEMU we changed two components of the virtualisation engine: Firstly, we extended the TLB refill function. Secondly, we modified the templates for *store* and *load* operations.

QEMU precisely emulates the behaviour of the hardware TLB and performs *TLB refill* each time QEMU's Tiny Code Generator accesses an unknown virtual address. We modified the MMU fault-handling routines[6] in such a way that they track accessed pages marked with the C-bit. Each TLB element inside QEMU includes a *tlb_address field* which contains the address of a memory page. The lowest bits of this field can be used to flag encryption (i.e., `TLB_VOLATILE`).

QEMU uses many techniques to improve performance. One of them is direct access to memory content without involvement of load/store functions. This technique is used for instructions, which do not involve PC registers and is called *fastpath*. For other operations, load/store templates are used and this is called *slowpath*. We disabled fastpath, thus all memory related operations use our customised load/store templates.

QEMU implements load and store functions with templates[7]. These templates, firstly, translate virtual addresses to physical addresses using a virtual TLB. Secondly, these templates perform actual loading/storing of data from/to resolved addresses. For each operation we check the lowest bits of the stored tlb_address, and perform encryption (store) or decryption (load) of the content, when the `TLB_VOLATILE` flag is set.

To alleviate testing and debugging, we are using byte-wise XOR (exclusive disjunction) as a cryptography algorithm instead of implementation of AES encryption. Byte-wise XOR does not affect the performance of QEMU, compared to AES, but still applies an encoding.

B. Sensitive data tracking

We modified the QEMU virtualisation platform to track accesses of encrypted memory. For a given application, as a first step, we tried to find a memory buffer, or a kernel subsystem which deals with the known sensitive data, for example, a plaintext password. For that, we made a snapshot of a virtual machine and parsed it. For each appearance of the sensitive data, we manually mark the memory region as encrypted, and then, enable the *encrypted memory tracker* inside QEMU, which was additionally developed. This tracker works as a hook for the load and store routines, and each time any code accesses encrypted memory, it retrieves the corresponding execution context (Figure 4). From this context, the tracker retrieves the `EIP` and `EBP` registers, recursively walks over the whole call trace and identifies all functions of

[4]mm/readahead.c: __do_page_cache_readahead()

[5]mm/filemap.c: grap_cache_page_write_begin()

[6]x86_cpu_handle_mmu_fault and tlb_set_page_with_attrs

[7]helper_le_ld_name and helper_le_st_name for little-endian architectures

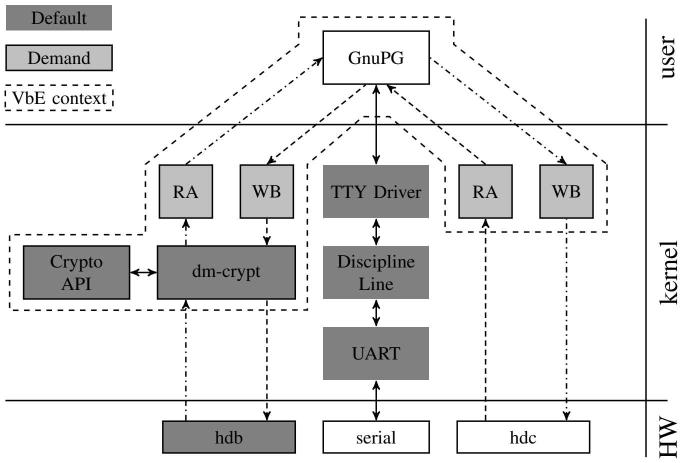


Fig. 5: SSDS of the use case

the trace by the retrieving the corresponding names from the kernel image. As a result, for each access to encrypted memory we received a call trace, analysed it, and modified the kernel accordingly by marking new regions from where/to which the data migrates. We repeated the procedure multiple times until we identified all SSDSs for chosen use cases.

VII. Evaluation

We considered the following use case scenario as a basis for our evaluation: A user receives an encrypted archive via email and saves it into non-encrypted storage. The archive is composed of encrypted files, which contain security sensitive information. The user decrypts the archive, edits files, re-encrypts them and sends a new archive to the sender. During these procedures, the Volatility infrastructure is used against data retrieval by an adversary.

A. Use of Security Sensitive Data Sets

Let us consider this example from the SSDS' point of view. Firstly, to prevent any leakage, the user needs to store decrypted files inside of encrypted storage. However, the original archive might be stored inside non-encrypted storage, since it is already encrypted in the first place. The second archive created from the modified files can also be stored on non-encrypted storage.

Firstly, the SSDS of the use case includes the necessary file I/O operations, such as the mounting of encrypted storage by (*cryptsetup*) and corresponding objects created inside the kernel. Secondly, the utility for encryption and decryption (*GnuPG*) should be protected by the encrypted memory use. Similarly to *cryptsetup*, this utility additionally requires encrypted input, which is used to read a passphrase from the console. Thirdly, an editor which is used to perform modification to the unpacked files should also be encrypted. Finally, SSDS of the use case includes additional objects created by the kernel during I/O operations, page caches (Figure 5, only *GnuPG* is depicted).

B. Development and testing infrastructure

Our development and testing infrastructure includes a whole systems stack: from the extended version of the QEMU emulator and an SME-enabled platform up to user space programs. We used the Yocto Project Build System version 16.0.1 for infrastructure building, deployment and testing. This infrastructure includes:

- QEMU virtualisation platform (2.7.0)
- Hardware platforms based on AMD EPYC 7281 with Micron 1100 SSD and 128 GiB DDR4 RAM
- Linux kernel (4.8.12), busybox (1.24.1), GCC (6.2.0)
- Applications and utilities: GnuPG (2.1.14), cryptsetup (1.7.2), hdparm (9.48)

We used QEMU to emulate persistent and encrypted platform and for detection of SSDSs (see Section V). We used the hardware platform for final benchmarks and the measurement of the impact of memory encryption.

C. Measurements

The goal of the evaluation is to demonstrate that the outlined complex use case can be protected by encrypted memory without sacrificing performance. For that, we used several benchmarks and applications.

a) Typing simulation: We developed a typing simulator. This simulator was written in Python and emulated the behaviour of an editor: insertion and removing of new words, movements within the document, saving of the file and more. We compared the time necessary to perform the same set of editing actions for vanilla and Volatility, but since we used a slow serial interface to communicate with the simulator it did not demonstrate any difference in the performance.

b) Performance of encrypted SSDSs: We also measured the performance of VbE system components involved in the use case. Firstly, we used the `hdparm` utility to measure read performance of encrypted and non-encrypted storage devices. Secondly, we used a simple `dd` between mounted encrypted and non-encrypted devices. Finally, we used GnuPG to encrypt a security-sensitive file. For the last benchmark, we used GnuPG for encryption of randomly-generated plan text files of varying sizes, starting from 1 MiB up to 300 MiB, and located inside different types of media, in accordance with the outlined scenario. We repeated our experiments several times with further averaging and compared performance of the same hardware platform with two different configurations, with and without of Volatility. Results of the benchmarks are presented in Table I.

TABLE I: Volatility benchmarks (MiB/s)

Setup	`hdparm` (non-enc)	`hdparm` (enc)	`dd`	GnuPG
Baseline	427.5	240.1	381.7	260.8
Volatility	427.5	238.7	379.9	260.2

As one can see, in all tests performance of the baseline is very close to the same of Volatility. In the case of non-encrypted `hdparm`, results are equal, since the Volatility components are not used for a non-encrypted media. For the remaining tests, the difference is less than 0.6%.

VIII. Related works

Various projects proposed low-level encryption techniques to protect against attacks based on physical access. For example, Aegis [16] and CryptoPage [17] encrypt and decrypt off-chip data transferred to physical memory. These works were mainly devoted to enable whole-system encryption and the effective implementation of memory encryption. Volatility targets selective encryption for systems featuring persistent main memory.

Cryptkeeper [18] proposed a more selective approach, where the whole memory is partitioned into two parts: plaintext and encrypted data. The most frequently used pages were not encrypted, while most infrequently used pages were encrypted. The goal of this technique is to reduce the amount of security sensitive data available in non-encrypted form stored in main memory. Volatility makes this general line of selective encryption much more strict. We never expose security sensitive data in a non-encrypted form and enable memory encryption on a per-page basis.

CPU-bound encryption is a software-based approach. The main feature of this kind of encryption is that the encryption keys are located inside the CPU and never leave it. Volatility is similar to these works targeting attack prevention, but does not provide or improve existing methods of memory encryption. While projects like PRIME [19], Armored [20], and copker [21] focus on the implementation of transparent memory encryption, Volatility targets selective use of encrypted memory to enable encryption of persistent systems.

Other works like TRESOR [22] and Loop-Amnesia [23] apply CPU-bound memory encryption to provide encrypted storages. These projects neither provide encrypted memory as a new system component, nor do they consider persistent systems as a target. In addition, transparent memory encryption, enabled by a Trusted Platform Module (TPM) (Hypnoguard [24]) or hypervisor (TreVisor [25]), does the same. In contrast, Volatility focuses on a holistic approach by supporting encrypted persistent memory in kernel and user spaces.

Ramcrypt, enables per-process memory encryption implemented via a sliding window [26]. Conceptually, the general idea of Volatility is close to Ramcrypt, but we encrypt not only the target processes, but all system entities involved into interaction with security sensitive data of target process. Moreover, the sliding window does not deal with persistence.

IX. Conclusion

In this paper, we present Volatility, a software system aimed at the protection of secrets in persistent systems from data leakage via attacks based on physical access. At the low level, Volatility provides mechanisms for allocation of VbE memory regions: memory regions transparently encrypted by the AMD SME. At the high level, it ensures the data protection over all relevant data flow paths without missing the persistence of the whole system.

Acknowledgement

This project was supported by the German Research Foundation (DFG) under priority program SPP2037 grant no. KA 3171/6-1.

References

[1] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable DRAM alternative,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[2] Y. Huai *et al.*, “Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects,” *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[3] Everspin, “MRAM into mainstream,” 2016.

[4] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: cold-boot attacks on encryption keys,” *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[5] D. Kaplan, J. Powell, and T. Woller, “AMD memory encryption,” Tech. Rep., 2016.

[6] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories,” *ACM Sigplan Notices*, vol. 47, no. 4, pp. 105–118, 2012.

[8] X. Li, K. Lu, X. Wang, and X. Zhou, “NV-process: a fault-tolerance process model based on non-volatile memory,” in *Proceedings of the Asia-Pacific Workshop on Systems*. ACM, 2012, p. 1.

[9] D. Narayanan and O. Hodson, “Whole-system persistence,” *ACM SIGARCH Computer Architecture News*, pp. 401–410, 2012.

[10] V. A. Sartakov and R. Kapitza, “NV-Hypervisor: Hypervisor-based persistence for virtual machines,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 654–659.

[11] V. A. Sartakov, A. Martens, and R. Kapitza, “Temporality a NVRAM-based virtualization platform,” in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2015, pp. 104–109.

[12] G. Heiser, E. Le Sueur, A. Danis, A. Budzynowski, T.-l. Salomie, and G. Alonso, “RapiLog: Reducing system complexity through verification,” in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 323–336.

[13] S. E. Hallyn and A. G. Morgan, “Linux capabilities: Making them work,” in *Linux Symposium*, vol. 8, 2008.

[14] A. Rubini and J. Corbet, *Linux device drivers*. " O’Reilly Media", 2001.

[15] R. Love, S. H. W. Are, A. C. Linus, and B. W. Begin, *Linux kernel development second edition*. Novell Press, 2005.

[16] G. E. Suh, C. W. O’Donnell, and S. Devadas, “Aegis: A single-chip secure processor,” *IEEE Design & Test of Computers*, 2007.

[17] G. Duc and R. Keryell, “Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection,” in *Computer Security Applications Conference, 2006. ACSAC’06. 22nd Annual*. IEEE, 2006, pp. 483–492.

[18] P. A. Peterson, “Cryptkeeper: Improving security with encrypted ram,” in *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*. IEEE, 2010, pp. 120–126.

[19] B. Garmany and T. Müller, “PRIME: private RSA infrastructure for memory-less encryption,” in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 149–158.

[20] J. Gotzfried and T. Muller, “ARMORED: cpu-bound encryption for android-driven ARM devices,” in *Availability, Reliability and Security (ARES), Eighth International Conference on*. IEEE, 2013, pp. 161–168.

[21] L. Guan, J. Lin, B. Luo, and J. Jing, “Copker: Computing with private keys without RAM,” in *NDSS*, 2014, pp. 23–26.

[22] T. Müller, F. C. Freiling, and A. Dewald, “TRESOR Runs Encryption Securely Outside RAM.” in *USENIX Security Symposium*, vol. 17, 2011.

[23] P. Simmons, “Security through amnesia: a software-based solution to the cold boot attack on disk encryption,” in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 73–82.

[24] L. Zhao and M. Mannan, “Hypnoguard: Protecting secrets across sleep-wake cycles,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 945–957.

[25] T. Müller, B. Taubmann, and F. C. Freiling, “TreVisor,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2012, pp. 66–83.

[26] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes, “Ramcrypt: Kernel-based address space encryption for user-mode processes,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 919–924.