Freeze & Crypt: Linux Kernel Support for Main Memory Encryption

Manuel Huber, Julian Horsch, Junaid Ali and Sascha Wessel
Fraunhofer AISEC, Garching near Munich, Germany

Keywords: Memory Encryption, Mobile Device Security, Data Confidentiality, Operating Systems Security.

Abstract: We present Freeze & Crypt, a framework for RAM encryption. Our goal is to protect the sensitive data the processes keep in RAM against memory attacks, such as coldboot, DMA, or JTAG attacks. This goal is of special significance when it comes to protect unattended or stolen devices, such as smartphones, tablets and laptops, against physical attackers. Freeze & Crypt makes use of the kernel's freezer, which allows freezing a group of processes by holding them firm in the so-called refrigerator. Inside, frozen processes inescapably rest at a point in kernel space where they cannot access their memory from user space. We extend the freezer to make arbitrary process groups transparently and dynamically encrypt their full memory space with a key only present during en- and decryption. When thawing a process group, each process decrypts its memory space, leaves the refrigerator and resumes normal execution. We develop a prototype and deploy it onto productively used mobile devices running Android containers. With this application scenario, we show how our mechanism protects the sensitive data in RAM against physical attackers when a container or device is not in active use.

1 INTRODUCTION

For decades now, the digital world has been finding its way deeper and deeper into our business and private lives. We leave behind the sensitive traces of our actions not only in the cloud, or on our persistent storage, but also in main memory. The data applications keep there, such as credentials, pictures, passwords, or key material, usually remains in plaintext (Apostolopoulos et al., 2013; Ntantogian et al., 2014; Pettersson, 2007; Tang et al., 2012). Especially in sensitive corporate or governmental domains, the reliable protection of valuable and possibly classified data is an important topic. Memory attacks, such as Joint Test Action Group (JTAG) (Weinmann, 2012), coldboot (Halderman et al., 2009; Huber et al., 2016; Müller and Spreitzenbarth, 2013), or Direct Memory Access (DMA) attacks (Becher et al., 2005; Boileau, 2006; Break & Enter, 2012; Devine and Vissian, 2009), have proven effective in disclosing secrets in RAM.

To tackle this problem, we present a framework for the Linux kernel, Freeze & Crypt (F&C), which encrypts sensitive data in RAM to solidly protect unattended devices or workstations. We design F&C to comply with multiple platforms, kernel versions and to incur only minimal changes sustaining the kernel's common operability. Our mechanism allows the transparent and selective protection of the full process space of arbitrary processes running on the system. We introduce F&C into the kernel building upon the existing control groups (cgroups) freezer subsystem. The cgroups mechanism allows the dynamic formation of groups of processes, also referred to as cgroups. The different cgroups subsystems enforce their own policies on cgroups in order to limit their access to system resources. The freezer subsystem, short freezer, allows the freezing and thawing of cgroups, i.e., their suspension and resumption.

When freezing a cgroup, the kernel sends a signal to the processes in the cgroup. Each of those processes reacts to that signal by entering the so-called refrigerator in kernel space. The refrigerator is a loop which ensures that processes neither execute in user space nor react to external events. When frozen, the user space part of a process has no means to access or alter its memory space. The refrigerator is thus a proper place to temporarily alter the processes' memory without side effects occuring on its user space part. In particular, we make the processes en- and decrypt the segments of their memory space, e.g., the heap, stack, code and anonymous segments, on their own and in parallel. This makes our approach especially on multi-core systems very efficient. Upon thawing a cgroup, the processes decrypt their memory before we allow them to leave. For the parallelization,

Huber, M., Horsch, J., Ali, J. and Wessel, S.
Freeze & Crypt: Linux Kernel Support for Main Memory Encryption.
DOI: 10.5220/0006378400170030
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, pages 17-30
ISBN: 978-989-758-259-2

F&C synchronizes the processes and threads inside the refrigerator, as they operate on shared resources. The key we use for en- and decryption of a cgroup is present only during freezing and thawing and may change on each freeze.

We demonstrate the utility of our mechanism with a working, real-life large scale application scenario in productive use, which is a virtualization platform running multiple Android containers on one smartphone (Andrus et al., 2011; Huber et al., 2015). With this scenario, we show how to employ our mechanism to protect the memory of Android containers against attackers with full access to the devices, for example, after theft. This platform is especially suitable, safely accounting for key management with a Secure Element (SE) to prevent brute-force attacks on the encryption key. In particular, our contributions are:

- A generic concept for efficient process memory en- and decryption in the kernel based on process groups.
- The development of a prototype and its integration into a complex mobile device architecture with key management to thwart physical attacks.
- The real-life application of the prototype on productively used smartphones with real users.
- A thorough security and performance evaluation to demonstrate the practical usability on mobile devices.

The paper is organized as follows. We first describe the threat model in 2 before presenting related work in 3. In 4, we present the design of F&C. We present an overview of the mobile device architecture and our extensions to leverage F&C in 5. Next, we elaborate the implementation of our prototype in 6. Our performance and security evaluations can be found in 7 and 8 before the conclusion in 9.

2 THREAT MODEL

We consider an attacker who aims at gaining sensitive data from an unattended and non-tampered device under protection of F&C. The attacker obtains physical access to the protected system, has sufficient time and the ability to access both volatile and persistent memory. For accessing the memory, the attacker exploits hardware and software vulnerabilities, such as through DMA, JTAG or coldboot attacks. The attacker is unable to execute evil maid attacks, i.e., to covertly deploy backdoors on the device waiting for the user to return. This implies that a device once tampered with is not trusted again, e.g., after theft or loss, or because the user notices the tampering attempt.

Regarding our application scenario, the adversary may be in possession of the SE, but lacks knowledge of the SE's passphrase and is unable to unveil the secrets stored on the SE. Furthermore, the attacker has no means to break cryptographic primitives.

3 RELATED WORK

In the following, we discuss previous work on memory protection. Several hardware-based memory encryption architectures, such as Aegis or the XOM memory architecture, have been proposed (Gutmann, 1999; Lie et al., 2003, 2000; Suh et al., 2007; Würstlein et al., 2016). Sensitive data of protected processes is left unencrypted solely in the processor chip, which is the single trusted component. These hardware architectures are difficult and expensive to realize and usually not found on consumer devices. There also exist processors with extensions to provide secure enclaves, which can be leveraged to thwart memory attacks, such as the ARM TrustZone, or Intel SGX. These enclaves constitute hardware-protected memory areas to which the OS can move sensitive data, but are currently still limited, e.g., regarding the size of the memory areas. Developers need to specifically design enclave-aware, hardware-dependent software and the underlying OS must support the processor extensions.

On the side of software-based implementations, some methods only protect a specific key, e.g., for Full Disk Encryption (FDE), in RAM from memory attacks. Approaches for x86 (Müller et al., 2010, 2011; Simmons, 2011), as well as for ARM (Götzfried and Müller, 2013) and hypervisors exist (Müller et al., 2012). The approaches either store the key in the CPU/GPU registers, or in the CPU cache, and implement the cipher on-chip at the cost of performance. These approaches leave all other assets in RAM unprotected and are hence vulnerable to memory attacks. Concepts protecting the process space are either runtime encryption techniques where process memory is encrypted throughout process runtime, or techniques which suspend processes in order to encrypt their memory, as is ours.

Cryptkeeper (Peterson, 2010) is an extension of the virtual memory manager to reduce the exposure of unencrypted data in RAM. The mechanism separates RAM into a smaller unencrypted working set of pages, called the *Clear*, and an encrypted area, the *Crypt*. By the time the Clear fills up, pages are automatically swapped into the Crypt and decrypted on demand. As a runtime encryption technique, there is a notable performance impact on the system and

the mechanism always keeps an undefined amount of RAM unencrypted. HyperCrypt (Götzfried et al., 2016a) and TransCrypt (Horsch et al., 2017) transfer this idea into a hypervisor to transparently encrypt the full memory of a guest OS.

RamCrypt (Götzfried et al., 2016b) is an encryption approach on x86 Linux that transparently encrypts the memory of running processes. For key-hiding, RamCrypt stores the memory protecting key in processor registers. Therefore, deep interference into the kernel's page fault handler is necessary to encrypt pages and to decrypt them when accessed. Processes to be protected have to be marked a priori by setting a flag inside the ELF program header. They encrypt anonymous segments only and there also remain unencrypted pages in a so-called sliding window. The cost of encrypting pages on the fly comes with a notable performance impact. Another problem common to runtime protection mechanisms is that a physical attacker gaining privileges on the system can simply request decryption of encrypted memory.

Hypnoguard (Zhao and Mannan, 2016) en-/decrypts memory during OS suspension/wakeup. The mechanism hooks into this procedure at stages where the OS is not active. At this point, there is no support for hardware devices, such as displays or keyboards. Therefore, their design requires to implement highly hardware-specific crypto routines for hardware accelerators and for drivers to interact with hardware devices, e.g., for passphrase input. The encryption key is bound to a TPM and encryption is executed in Intel's TXT environment. This makes the approach highly hardware specific and particularly cumbersome for portability and when interacting with different hardware devices. The approach in (Huber et al., 2017) focuses on the same goal, but constitutes a hardware-independent kernel mechanism, which can be easily integrated into Linux systems.

Transient Authentication (Corner and Noble, 2003) is also an approach for x86 platforms to protect processes transparently, but depends on the presence of a hardware token. The token provides fresh cryptographic keys. The concept comes with two protection variants, the *application-transparent* and *application-aware* protection. In the first mode, the system suspends and encrypts in-memory pages when the user removes the token. The only processes that remain running are tasks for transient authentication and OS threads. In application-aware mode, the developer can protect specific applications by utilizing a special API. This allows to selectively protect chosen assets, such as an application's secret key. However, this requires the modification of existing applications.

Approaches specifically tailored to the mobile domain have also been developed. Sentry (Colp et al., 2015) presents a concept for memory protection on Android devices. The user has to mark sensitive apps and OS subsystems in the settings menu. When the device gets locked, the mechanism encrypts specific memory of the chosen apps. For apps that run while locked, Sentry reads encrypted memory pages from RAM, decrypts and keeps them inside the ARM SoC with cache line locking (Chen et al., 2008). Pages are encrypted on a page-out before writing them back to RAM. With increasing background activity and a full cache, the performance strongly degrades since the mapping of called-in pages to the cache triggers costly page faults. The approach uses ARM specific (legacy) mechanisms. Hence, the feasibility strongly depends on the architecture and platform specific hardware features. Their prototype on the mobile device does not support cache locking and hence no runtime memory encryption.

CleanOS (Tang et al., 2012) is a mechanism integrated into the Android framework. This approach only works in combination with trusted, cloud-based services for key management to which the phone needs connectivity. The *idle eviction* mechanism encrypts data that is not in active use. Afterwards, the key is purged on the device and fetched on-demand from the cloud. The main modifications were made by the introduction of Sensitive Data Objects (SDOs), which represent sensitive user data, and a special garbage collector, eiGC. The latter searches and encrypts SDOs that were not used for a specific period of time. Apps either implement an SDO API to add and register SDOs, or the framework registers default SDOs along with heuristics to identify SDOs. Not only regarding key retrieval and availability concerns, but also due to the heuristics and workload to adjust apps, this mechanism represent no reliable scheme.

4 MEMORY PROTECTION CONCEPT

We first present an overview on F&C's overall design and the involved OS components in 4.1. Subsequently, we elaborate the synchronization of the processes during main memory encryption in 4.2. In the following, the term encryption also applies for decryption and we differentiate only where decryption differs from encryption.

4.1 Design of Freeze & Crypt

Based on the Linux kernel's cgroups freezer, F&C allows the dynamic creation of cgroups containing

the processes and threads whose memory space we encrypt. From here on, we subsume processes and threads under the term *task* and only differentiate when relevant. For every cgroup, we use an arbitrary key for each encryption pass. We differentiate memory segments upon the following categorization:

text. The executable, read-only code of a task.

data. Writable, as well as read-only data belonging to the task's code.

bss. Zero-initialized, writable data.

heap. A task's dynamically allocated memory.

stack. Local variables of functions and their parameters, etc.

anonymous mappings. Large, dynamic memory allocations a task can map into its process space and share with others.

file mappings. Files that are mapped to memory. For example, large writable files, but also read-only files, such as shared libraries. We further make it possible to choose between regular and non-regular mappings, as well as between their access permissions. Non-regular mappings are, e.g., memory-mapped devices, DMA or Inter-Process Communication (IPC) resources.

special mappings. The *timers*, *vectors*, *vdso* and *vsyscall* segments. They contain no confidential user data. The kernel manages these segments as system-wide shared kernel resources providing them to tasks as kernel interfaces.

Except for special mappings, a task can load or store confidential information in all other segments. F&C thus allows to select all other segments for encryption except the special mappings. Depending on the use case, fewer segments can be critical and thus omitted for encryption. We outline the critical segments for our use case in 5.

4.1.1 States of a Cgroup

1 depicts the four different states of a cgroup along with its tasks. On the system, we have an arbitrary number of cgroups in any of the states, as well as non-protected tasks which can potentially become part of a freezer cgroup or form a new cgroup. F&C allows us to create disjoint cgroups, to freeze or thaw multiple cgroups at the same time, and to assign different keys.

Thawed. In the default state, multiple tasks are *running*, such as $Task_1$ in 1, and the cgroup holds no key. The cgroups mechanism ensures the inclusion of future child tasks to the cgroup by default. The arrows demonstrate that tasks execute their code in user space, access their memory segments and possibly jump between kernel and user space when making system calls.

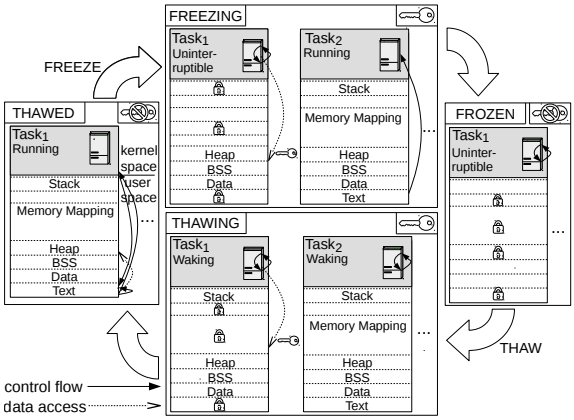

Figure 1: The four states of the freezer along with the behavior of tasks in the different states.

Freezing. After starting the freeze, the cgroup goes into the state *freezing* and has an arbitrary encryption key assigned. Tasks in the refrigerator go into an *uninterruptible* state and dwell inside until the cgroup is thawed. $Task_1$ already entered the refrigerator and is currently encrypting its memory segments. The arrow for the running $Task_2$ shows that it is just jumping from its execution in user space to kernel space into the refrigerator. In the refrigerator, the user space part of a process has no means to access or alter its memory. The tasks are agnostic of incoming IPC or external events, such as kill signals, which they process after thawing.

Frozen. After finishing the encryption, the cgroup is *frozen* and does not hold any key. Its tasks are all in the *uninterruptible* state, stuck in the refrigerator in kernel space off the run-queue. Once scheduled, the tasks immediately call the scheduler again to switch to another task. All tasks have encrypted the previously specified memory segments.

Thawing. After starting the thaw, the cgroup is in the state *thawing* and decrypts with the same key as used for freezing. The cgroup brings its tasks into a *waking* state. When scheduled, the tasks simultaneously decrypt their segments before leaving the refrigerator. $Task_1$ is about to decrypt its segments, while $Task_2$ has already finished. We retain $Task_2$ in the refrigerator for synchronization until all tasks of its cgroup finish decryption. This causes depending tasks to be released simultaneously.

4.1.2 Extension of the Freezer

We extend the freezer's initialization, freezing and thawing functionalities to manage the protection of cgroups. 2 illustrates these functionalities with a se-

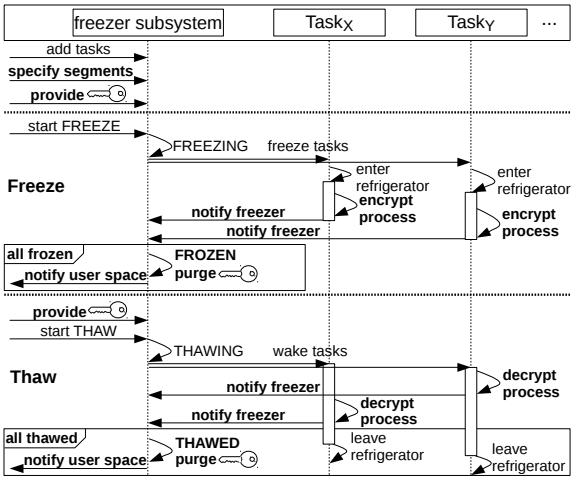

Figure 2: Extensions (bold) to the initialization, freezing and thawing procedures of a cgroup with its tasks in F&C.

quence diagram. Bold elements stand for the new functionality we introduce to the freezer.

Initialization of F&C. Privileged processes in user space can manage the cgroups functionality via an interface to the kernel, which allows for creating cgroups and adding or removing tasks. To initialize a freezer cgroup, a privileged process adds tasks to the cgroup, here $Task_X$ and $Task_Y$. We extend the interface by the option to associate the freezer with an arbitrary key and the list of memory segments to be encrypted. The concept intentionally leaves the key management open to the specific use case F&C gets applied. The key can, for example, originate from password derivation, a TPM or an SE, as described in our application scenario in 5.

Freezing Procedure. As depicted in 2, a privileged process starts the freeze and thaw of cgroups. The freezer changes its state and signals its tasks to enter the refrigerator. Every task notifies the freezer after finishing its encryption. When the freezer has received all encryption notifications, the cgroup goes into the state *frozen* and the freezer purges the encryption key. To erase potentially remaining sensitive remnants, the freezer also zeroes out pages freed by running or terminated processes, as well as the used cipher kernel structures and the relevant kernel stack range in memory after the encryption. Finally, the freezer notifies user space about the terminated encryption of the cgroup. The standard freezer only reveals and becomes aware of its state when user space requests the actual state. In F&C, the subsystem actively manages its state to be able to purge the key and other remnants as early as possible. We also actively notify user space, e.g., entities managing the cgroups encryption, about the change of state. We show the usefulness of this feature in our application scenario.

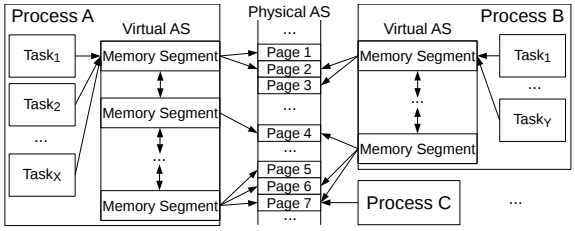

Figure 3: Relationship of processes and threads, their memory mappings and sharing of resources.

Thawing Procedure. Before triggering the thawing of a cgroup and therefore starting the decryption, a privileged process passes the corresponding key to the freezer. The freezer wakes every included task. When scheduled, this normally causes a task to leave the refrigerator. In our mechanism, the tasks first decrypt their previously encrypted memory segments and notify the freezer. Unlike the original freezer, we do not immediately switch to the state *thawed*, but wait until the last task finishes decryption and notifies the freezer before switching states. Then, the tasks safely leave the refrigerator when scheduled and resume their execution. The freezer purges the decryption key, other remnants and notifies user space. The key is hence only present in the freezer during en- and decryption.

4.2 Synchronization of Tasks

Since the tasks of a freezing cgroup enter the refrigerator in parallel and share resources, we synchronize their concurrent encryption. The kernel's memory management is responsible for resource sharing. Shared resources can be Address Spaces (ASs) and physical pages, as shown in 3. The virtual AS consists of the different memory segments referring to physical pages in memory. Virtual ASs, i.e., processes, can share physical pages. In the following, we first focus on the shared ASs before addressing shared pages.

4.2.1 Shared Address Spaces

A task can spawn threads and fork new processes. The kernel assigns a forked process its own AS. When spawning, the parent process shares its AS with the spawned thread. 3 illustrates the sharing of ASs for the tasks of process A and B. Our goal is to ensure that every AS is encrypted by exactly one task to avoid multiple encryptions of the same AS. Without synchronization, sharing tasks would encrypt the same AS repeatedly when entering the refrigerator.

4 focuses on the chronological sequence tasks run through in the refrigerator with process A and B belonging to the same cgroup. Upon freezing, the tasks

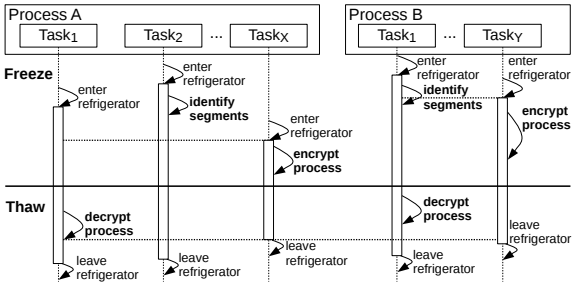

Figure 4: Synchronization of tasks entering and leaving the refrigerator during freeze and thaw.

enter the refrigerator at an undefined point in time in an order determined by the scheduler. The first task to enter the refrigerator, here $Task_2$ of process A and $Task_1$ of process B, predetermines the segments to be encrypted. The last entering tasks, $Task_X$ and $Task_Y$ respectively, encrypt the previously identified memory segments. Only the last task can safely encrypt the segments, because at that time all other tasks sharing the AS have entered the refrigerator. The first task thus identifies the segments for the last arriving task to earliest possible start encryption. We ensure that encryption does not start unless all segments are identified. As depicted in 4, $Task_Y$ of process B encrypts its AS in parallel to $Task_X$. In reverse, only the first scheduled, waked task of a process decrypts the segments. Once all tasks have finished decryption, we allow them to leave the refrigerator.

4.2.2 Shared Physical Pages

The tasks use virtual addresses during their encryption. However, the physical pages effectively encrypted can be shared between process boundaries complicating the encryption process. Hence, we have to determine whether the page to be encrypted is shared before encrypting it in order to prevent the corruption of other ASs. Shared pages are contained in more than one AS and can be further categorized. First, these can simply be pages intended to be read-only for all tasks. Second, these can be writable Copy-On-Write (COW) pages, which are shared between processes for the time they only read the page. Third, shared pages can represent shared memory for IPC, where distinct processes work on the same set of physical pages. Fourth, pages could have been merged via the kernel's Kernel Samepage Merging (KSM) mechanism, which searches identical pages in process space and merges those pages to one shared page to save memory. Before a task encrypts a shared page, we ensure that the page is not referenced by other tasks possibly not part of the cgroup or not yet in the refrigerator. Our mechanism thereby ensures encryptingthese shared pages only once and not duplicating them by COW. When thawing, an encrypted page can always be decrypted as long as no other task is currently decrypting it.

5 APPLICATION SCENARIO

We demonstrate the capability of F&C on mobile devices in practical use. We use Nexus 5 smartphones running multiple Android containers based on a virtualization platform (Andrus et al., 2011; Huber et al., 2015; Wessel et al., 2015). When the mobile device is actively used, one Android container is in foreground, while one or more containers run in background. We encrypt background containers and only leave actively used containers unencrypted. After the device is idle for a certain period, we ensure to encrypt all containers in order to protect all sensitive data from memory attacks. Like on common smartphones, we require users to re-authenticate in order to start or resume a suspended, encrypted container. The architecture suits our mechanism, because it builds on an SE with two-factor authentication for secure cryptographic key management, e.g., a smartcard via NFC or a microSD. Using an SE prevents brute force attacks on the RAM encryption keys, but can weaken usability when re-authenticating. However, the usage of an SE was the preferred choice due to the high security requirements for our productive scenario. For other use cases, our system can easily be adapted to a less secure, but more usable scheme, such as PIN or password based key derivations or swipe patterns.

We first describe relevant components of the virtualization platform and our extensions to leverage F&C in 5.1. Then, we describe how we employ F&C to protect the containers' sensitive data in memory in 5.2. 5.3 details how we handle background events for frozen containers, such as incoming phone calls or alarms.

5.1 Components and Extension of the Virtualization Platform

5 shows the relevant components of the virtualization platform, including our extensions. The illustration depicts a scenario with running containers $C_{0..X}$ on top of trusted components in user and kernel space. $C_{1..X}$ are isolated Android user containers with apps, for example, a private and a business container. We assume that incoming sensitive data for $C_{1..X}$ is encrypted and that encryption terminates inside $C_{1..X}$. C_0 is a minimal, hardened and trusted management container providing functionality to virtualize hard-

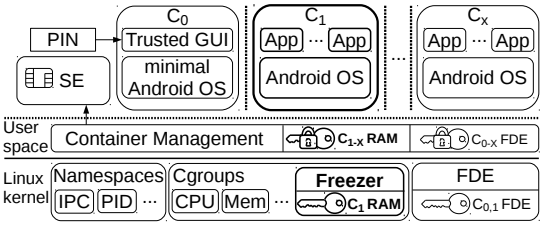

Figure 5: Most relevant components and extension of the virtualization platform for mobile device.

ware devices for $C_{1..X}$ and not containing any sensitive user data. Hence, we do not encrypt C_0's RAM contents. C_0 has a Trusted GUI app where the user provides the SE's passphrase (PIN).

$C_{0..X}$ are embedded into separate namespaces on kernel level. Linux namespaces provide processes within a namespace with their own view on the system's resources, such as IPC, PID, or network namespaces. This isolates processes from different namespaces making them unaware of other namespaces and IPC takes place through well-defined channels only (Huber et al., 2015). The Container Management (CM) assigns each started $C_{0..X}$ to a different cgroup, coherent to the distinct namespaces. The platform uses the CPU, memory and devices cgroups subsystems to regulate the containers' access to hardware resources. In user space, the CM manages the configuration, startup and switching of containers. Containers store their persistent data in images protected with kernel-based FDE. Therefore, the platform uses wrapped, persistent disk encryption keys *$C_{1..X}$ FDE* that can only be unwrapped using the SE.

We extend the CM with functionality to configure F&C and to generate, wrap and unwrap the RAM encryption keys *$C_{1..X}$ RAM* using the SE. 5 depicts the scenario where C_1, in bold face, is in foreground, while the other containers are in background. The illustration shows the freezer using unwrapped key *C_1 RAM*, which indicates that the system has locked C_1, is about to switch to C_0 and encrypting C_1 in background. Meanwhile, the FDE module solely keeps the unwrapped keys *$C_{0,1}$ FDE*. The kernel does not require the keys *$C_{2..X}$ FDE*, because $C_{2..X}$ are frozen and cannot access their filesystem. After freezing C_1 completes, the CM also removes C_1's key *C_1 FDE* from the kernel. The scenario allows for omitting containers from RAM encryption, e.g., containers without corporate secrets.

5.2 Container Protection and Key Management

For the encryption of $C_{1..x}$, we consider all selectable memory segments described in 4.1 relevant for our

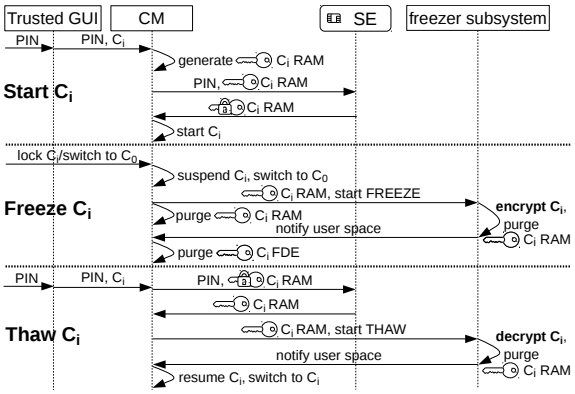

Figure 6: Chronological sequence for starting, freezing and thawing containers along with key management operations.

scenario except the following:

Read-only Executable File-mapped Segments. This is shared library code, which does not contain sensitive user data in our system, but solely shared constant data. Since each container features a full userland, $C_{1..x}$ include separate libraries, e.g., libc. The data segments of libraries do not fall in this category and are hence encrypted.

Writable Non-regular File-mapped Segments. These segments represent memory areas of memory-mapped devices, IPC resources, or DMA files where drivers share memory with hardware devices. Encrypting such memory can corrupt the memory space on most platforms, because, for example, hardware devices are not aware of the encryption.

All other segments must be protected, because processes might store or load sensitive data there. The diagram in 6 describes the procedure for protecting $C_{1..X}$'s sensitive data. The illustration shows the steps of starting, encrypting and decrypting C_i.

Start C_i. To start C_i, we enter the PIN of the present SE in the Trusted GUI of foreground C_0. The GUI passes the information to start C_i with the PIN to the CM. The CM randomly generates a fresh key *C_i RAM* using the kernel random number generator seeded with hardware entropy based on hardware random numbers. The CM unlocks the SE and uses it to wrap the generated key *C_i RAM*. At this point, the CM holds both the wrapped and unwrapped key *C_i RAM*. The unwrapped key is stored to be able to immediately encrypt C_i without user interaction and without the SE when locking or switching back to C_0. Reversely, the wrapped key is stored for the next decryption pass of C_i using the SE. Next, the CM starts C_i in foreground, which brings C_0 to background. Therefore, the CM creates C_i's namespaces, configures its cgroups and mounts its images. The CM also speci-

fies the segments to be encrypted. Starting C_i includes unwrapping C_i's disk encryption key *C_i FDE* using the SE to provide the key to the FDE module for accessing the encrypted data image, omitted in 6.

Freeze C_i. We freeze and hence encrypt C_i's RAM when the user actively switches back to C_0, or when the user or the system locks C_i, causing a switch back to C_0. According to 6, the CM then suspends C_i and immediately switches to C_0. The CM provides the freezer with the stored, unwrapped key *C_i RAM* and starts C_i's freeze. This triggers F&C, which encrypts C_i. In the meantime, the CM purges its stored, unwrapped key *C_i RAM* from user space. After the encryption, the freezer purges its utilized key *C_i RAM* in the kernel and notifies the CM. The CM then purges the no longer required, unwrapped key *C_i FDE* in user and kernel space. We destroy the encryption keys by overwriting them in RAM and by flushing the corresponding caches. After that, the CM stores only the wrapped RAM and disk encryption keys, leaving no trace of C_i's volatile and persistent data. All confidential assets in kernel memory are deleted and the non-encrypted segments contain no confidential information. In other use cases, the entity managing the encryption can purge further assets in kernel at this point if present, e.g., IPSec credentials (a kernel level VPN mode).

Thaw C_i. Only to start and to thaw C_i, i.e., when decrypting and putting C_i to foreground, the user has to provide the PIN of the SE to the Trusted GUI. The CM unwraps the wrapped key *C_i RAM* using the SE and provides the freezer with the unwrapped key *C_i RAM*. Afterwards, the CM triggers the thaw. The freezer decrypts C_i, purges its utilized key in the kernel and notifies the CM. Then, the CM resumes C_i, switching it to foreground. To simplify matters, 6 omits that the CM also unwraps the wrapped key *C_i FDE* and provides it to the FDE module before thawing C_i. The CM keeps its user space copy of the unwrapped key *C_i RAM* for the next freeze. At this point, the CM could also generate a new unwrapped key *C_i RAM* with a wrapped counterpart for the next freeze. This would prevent replay attacks swapping old portions of encrypted RAM. However, such scenarios are not part of our threat model and we consider this threat negligible.

5.3 Background Activities

The CPU sleeps on suspended smartphones. For background activities, devices sustain their connection to external sources via hardware components and via interrupt controllers. In case of an event, e.g., causing a notification, the CPU and hence processes get woken. In our case, events for encrypted containers are similarly processed by the kernel and due to the hardware device virtualization, events are forwarded to the virtualization infrastructure. This enables us to handle background activities for frozen C_i.

Hardware devices are either virtualized in user space or in the kernel on our platform. User space virtualized devices, such as the radio interface, have multiplexers in C_0. We thus receive incoming short messages and phone calls for possibly frozen C_i via C_0. We extend the architecture by adding functionality to notify the user of events for frozen C_i in C_0 using Android notification intents. After thawing the container, the user is able to take the call and to receive the message right after thawing. The same holds for devices virtualized in the kernel, such as the alarm or networking functionality. We track expired alarms and incoming network traffic for frozen C_i in the kernel and raise a notification in C_0.

In practice, only the notification handling for incoming network traffic is confined on our prototype. Apps mainly use the Google Cloud Messaging (GCM) infrastructure and receive data, such as instant messages, only when the system reports back to GCM. After thawing, the intended function of the apps however continues seamlessly and apps quickly receive their data. For full network notification handling, we could virtualize the connection management for C_i in C_0. Another possibility is to use our backend to which the CM connects. The backend then notifies the device and handles the remote connections, e.g., to the GCM services. The trusted connection would have to be established on behalf of C_i in C_0 or in the backend. Our current solution keeps the used connection credentials safe inside frozen C_i.

6 IMPLEMENTATION

In this section, we describe the implementation of our prototype for the Linux 3.4 up to recent ARM and x86 kernel versions. We start with the management of F&C from user space. Afterwards, we focus on the extensions we make to memory management and on the synchronization of tasks during freezing and thawing. Then, we elaborate the procedure used by a task to encrypt its memory space.

6.1 Interaction with User Space

In order to pass the key and the list of segments from user space to the kernel, we create additional files in the cgroups virtual filesystem. We read the key and list of segments we receive from user space via file

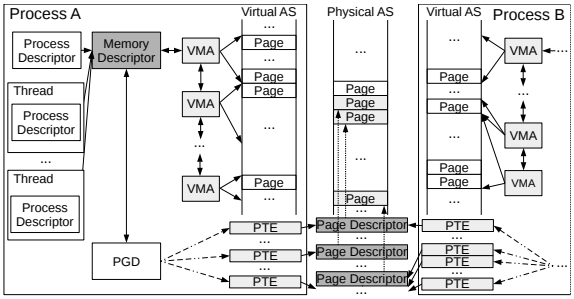

Figure 7: Linux memory management data structures and modifications highlighted in gray colors.

handles similar to the existing freezer state change functionality. However, getting notified in user space about changes in filesystems (`inotify`) does not work with the cgroups virtual filesystem's pseudo files. Instead, user space must periodically read the state file to check whether the freezer changed state. To avoid this polling, we use `eventfd` and the cgroups notification API to explicitly notify user space when freezing and thawing terminates.

6.2 Memory Management

A process and its spawned threads share the kernel structures which describe a task's memory layout. 7 shows relevant components the Linux kernel uses for memory management and how the tasks share resources. The illustration outlines the sharing of resources on the example of process A and B. Each task has its own process descriptor for process management which contains, e.g., the name and PID. We temporarily modify the light gray colored components during encryption, like Virtual Memory Areas (VMAs) and Page Table Entrys (PTEs). Medium-gray colors denote components to which we newly introduce functionality for handling shared resources.

The process descriptor points to a memory descriptor, which describes the virtual memory layout of a process. The memory descriptor tracks the number of users it has, i.e., the number of threads that share the memory descriptor with the parent process. We extend the memory descriptor to additionally count the number of frozen sharing tasks which are tasks that have already entered the refrigerator. This way, the tasks are able to determine whether they are the first or last one to enter the refrigerator. The memory the descriptor points to is reflected by a linked list of VMAs representing the memory segments we selectively encrypt. For the encryption, we make non-writable VMAs temporarily writable.

The memory descriptor keeps a reference to a Page Global Directory (PGD) used for translating a process' virtual addresses to corresponding physical addresses. Based on a page walk from the PGD traversing the Page Upper Directory (PUD) and Page Middle Directory (PMD), we determine the PTE corresponding to a virtual address if mapped, i.e., if the page exists in main memory. A PTE has several values tracking the page's state, e.g., if the page is shared between processes, or if the process has accessed or written the page. By writing a physical page, the kernel also possibly modifies the PTE's values. Before the encryption of a physical page, we thus save the PTE's values and restore them right after the page's encryption. This especially prevents file-backed pages accidentally being made persistent through the page cache by not setting the dirty flag.

A present PTE maps one physical page by referencing a so-called page descriptor, which describes one specific physical page in memory and which directly points to the page's physical address. We set a flag in the page descriptor indicating a lock on physical pages in memory during encryption. By locking the page, we make sure that the locking task obtains exclusive page access. Furthermore, we extend the page descriptor with the functionality to mark a page as encrypted. With this functionality and page locking, we ensure that tasks encrypt pages only once. To keep track of the entities referencing the page, the page descriptor holds a reverse map to referencing VMAs. Since VMAs have a back reference to their memory descriptor, we are aware of all tasks, possibly outside the cgroup, referencing the page described. This allows us to determine whether the shared pages a task references may be encrypted or not.

6.3 Synchronization of Tasks

A task entering the refrigerator first saves its current task state. We then increment a cgroup-wide barrier we newly introduce, which counts the tasks entering the refrigerator. When thawing, the barrier forces thawing tasks to wait in the refrigerator until decryption of the whole cgroup completes. In the next step, we increment the frozen user count of the task's memory descriptor. The first user, i.e., the first task, identifies the memory segments in the process space, which were selected for encryption. After possibly identifying the memory segments, a task goes as usual to the state `TASK_UNINTERRUPTIBLE`. If the process is not yet encrypted, the task checks if it is the last entering user of the memory descriptor. If yes, the task marks the memory descriptor as encrypted and encrypts the identified memory segments. After that, or in cases where the task was not the last user of the memory descriptor, the task notifies the freezer about being frozen. Then, the common freezer procedure executes.

The task flags itself FROZEN and ends up in a loop, checking if its cgroup is frozen or not. When scheduled, the frozen task executes the common functionality and indicates the scheduler to switch to another task.

When thawing a cgroup, the freezer wakes the cgroup's tasks, leaves the frozen state, and each task clears its FROZEN flag. The tasks normally leave the refrigerator and restore their previous process state. However, our mechanism handles the decryption of the tasks before leaving. For this purpose, thawing tasks first decrement the frozen user counter. This ensures that only the first thawing task using the memory descriptor decrypts the associated memory space. Otherwise, the task skips decryption. In the next step, every task decrements the cgroup-wide barrier. The last task about to leave the refrigerator completes the barrier. All other tasks waiting at the barrier are finally free to leave the refrigerator. The last task also notifies the freezer about the terminated decryption of the whole cgroup.

6.4 Process Space Encryption

For the encryption in main memory with the kernel crypto API, we apply the asynchronous bit-sliced AES CTR implementation using NEON instructions with a 256 bit key size. The CTR mode achieves especially high performance on multi-core systems through its parallelization. We use the physical page addresses as Initialization Vectors (IVs), resulting in distinct IVs for each encrypted block. During the encryption, a task iterates over the previously identified VMAs of its memory descriptor. The task first checks the VMA's write permissions. If a VMA is not writable, the task makes the VMA writable. The task then encrypts the VMA page by page. After encrypting the whole VMA, the task restores the VMA's write permissions, if necessary.

The page level encryption procedure starts by checking if the physical page to be encrypted is present in main memory. We skip non-present pages. We hence do not encrypt swap, as swapped pages are considered non-present. For the encryption of swap, we refer to standard Linux swap encryption. On a present page, the task tries to get our lock. Failing indicates that the page is already being encrypted by another task. Hence, the task skips the encryption of this page. The next step differs depending on whether we are in the en- or decryption process. Unlike in the decryption case where we can continue to decrypt a page right away, we are obliged to make sure the page is ready for encryption. We first check if the page descriptor was already marked as encrypted. In this case, the task releases the lock and skips the page. If the page was not already encrypted, the task checks if the page descriptor is referenced more than once. Multiple references indicate that the page is shared. If there is only one PTE referring to that page, the task immediately considers the page ready for encryption. Otherwise, the task must specifically check the page's readiness for encryption, because it is shared across AS boundaries. Another task of the cgroup considering that page for encryption will ensure the page's encryption later if the page is exclusively used by the cgroup. Simply encrypting a shared page without further inquiries would cause COW, which replicates that page and encrypts only the copy. On the other hand, preventing the page fault triggering COW would corrupt the AS of other processes if the page is not ready. The task releases the lock on the page if the page is not ready and skips it.

In case the page is ready, the task marks it as encrypted. Before writing the page, the task stores the PTE's flags. These flags indicate whether the page is, e.g., writable, dirty, or young. The task checks whether the page is writable or not, because writing a read-only page implies COW in the triggered page fault. Since the task made sure the page is only used within the cgroup, it circumvents the page fault and makes the PTE writable before encryption. In the next step, the task encrypts the physical page and subsequently restores the PTE's flags. This ensures that PTE flags remain unaltered. The task finally unlocks the page.

7 PERFORMANCE AND STATISTICS

In this section, we present our performance results and statistics on the tasks, VMAs, and pages encrypted with F&C in our application scenario. We configured the platform to run two full-fledged Android 5.1.1 containers in addition to the management container on top of a single Linux 3.4 kernel (4 KB page size; as on stock Android, no swap or KSM). This resembles a realistic scenario with a private and business domain. For the evaluation, we intentionally encrypt the both containers to challenge F&C with multiple, large cgroups. For common use, encrypting only the business container processing corporate secrets may be a more appropriate choice with higher usability. We adhered to the selection of segments for encryption as described in 5.1.

We identified six test users to productively utilize F&C for one working week on their commonly used Nexus 5 smartphones running the virtualization archi-

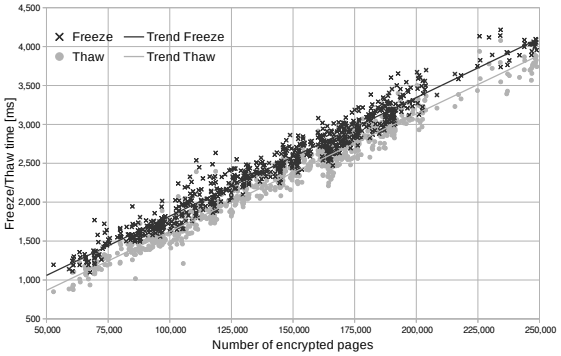

Figure 8: Performance of container freezes (crosses) and thaws (circles) in relation to the number of encrypted pages.

tecture to generate our statistics (Quad-core 2.3 GHz Krait 400 CPU). To deploy F&C on the phones, we provided chosen users a remote update with our modified kernel and virtualization layer. The users were able to seamlessly continue utilizing their two containers with their data and dozens of apps, e.g., social, business, mail, or media apps. Thawed containers continued running stable, even after a long freeze. The evaluation period also incorporated a number of especially challenging test cases with background events, where users stressed the memory limit or received phone calls in a frozen container. With these test cases, we verified that F&C reliably works even when the system is under heavy load, that the phone is still practically usable and that it remains fully stable.

To gather the statistics, we introduced a performance profiling mechanism to F&C. We measured the most important figures in the kernel during encryption, such as the duration of the encryption or the number of encrypted pages. 8 illustrates the measured en- and decryption times in relation to the number of pages. The black crosses and gray circles refer to container freeze and thaw times in milliseconds. In total, we have more than 730 container en- and decryption samples where our users run arbitrarily many apps on the smartphone for an indefinite period of time. The more apps run, the more pages are present, the more pages we encrypt. On average, about 142,500 pages were encrypted in parallel on the four available cores requiring 2,468 ms for freezing and 2,266 ms for thawing. In extreme cases where the Android low memory killer is triggered active and a container exhausts all of its resources, the duration increases roughly to fairly, still practicable, 4,000 ms. When starting the freeze of C_i, the CM directly and quickly switches back to C_0 (Huber et al., 2015). This means that a user intending to switch to another container C_j can start entering the SE's passphrase in C_0 during C_i's encryption in background. Since typing takes some time, the additional cost of a switch usually amounts to only the time required for thawing C_j. Both trendlines in 8 point out the linear growth in encryption times and show that decryption is faster than encryption. This is reasonable due to the additional synchronization effort during encryption. The variance originates from the many different events and processes that the kernel schedules in this complex system. The measured encryption time comprises the instant of writing to the freezer state file until user space receives the notification from the freezer. In between, the performance mainly depends on the synchronization inside the cgroup and on the scheduler. In sum, our solution is suitable for daily use in environments where protection of sensitive data plays a crucial role.

1 shows further statistics. On an average encryption cycle, 1,043 tasks were running, including 52 processes spawning 991 threads. This points out the large size of the cgroups and that users were running many apps. The second row shows that during freezing, the 52 encrypting tasks identified 27,410 VMAs, where 13,391 were subject to encryption. The remaining 14,646 VMAs represent the non-sensitive segments. Freezing tasks altogether had about 380,500 virtual pages mapped on average. When multiple PTEs point to the same physical page, there are numerous duplicate pages. This is why the total number of encrypted physical pages, about 142,500, is on average clearly smaller and many pages, about 238,000, could be skipped. The average number of encrypted pages accounts for about 560 MB of memory. The bottom row shows the classification of the encrypted pages into the different VMAs. Most pages are either part of file-backed or anonymous segments. Only a small number of pages belong to the stack, heap, data and text segments. A future improvement would thus be to zero out and unmap file-backed segments instead of encrypting them. This comes later at the cost of runtime performance, but is probably only hardly perceptible and can clearly improve suspension and resumption.

To measure the effects of F&C on power consumption, we separately conducted power measurements. The additional battery drain depends on the frequency of suspending and resuming containers. During the power measurements in the kernel, we automatically switched between the two user containers and the management container, suspending and resuming in 10s intervals for a 100 times. We independently repeated that experiment 5 times with charged phones, varying started applications and used a fixed key (to omit user-interaction). We measured a battery drain of about 2.5% without and no more than 4.0% with F&C. There is hence little perceptible effect on

Table 1: Statistics on the average number of tasks, VMAs, pages and the types of encrypted pages.

Tasks Total	Threads	Processes	
1,043	991	52	
VMAs Total	Skipped	Encrypted	
27,410	14,019	13,391	
Pages Total	Skipped	Encrypted	
380,545	238,090	142,455	
Pages Encrypted	File-backed	Anonymous	Text/Data/ Stack/Heap
142,455	71,606	68,976	1,873

power consumption with normal phone use.

8 SECURITY EVALUATION

We first discuss the security aspects of F&C itself and then extend the considerations to our application scenario. According to the threat model, we assume that the attacker gains full access to all volatile memory. This enables the attacker to obtain and analyze a full memory dump with the encrypted sensitive data. F&C encrypts the pages of the selected memory segments in RAM using AES in CTR mode and uses its unique physical address as IV for each page encryption. Identical pages are consequently encrypted with a different outcome and since the attacker cannot break cryptographic primitives, encrypted pages reveal no sensitive information. After encrypting a cgroup, we purge the key used in the freezer, as well as other AES remnants and freed pages. When the key is removed in user space as well, the adversary has no means to decrypt protected pages. This means that the analyst can only attempt to obtain sensitive data in non-encrypted pages and segments. F&C allows the selection of all memory segments for encryption, except the special segments containing no confidential information. For encrypted segments, we leave only those pages unencrypted which are shared with unencrypted, running processes. When making sure that processes which share sensitive data with other processes are contained in the frozen cgroups, we leave no sensitive pages unencrypted.

Regarding our application scenario, the goal of the attacker is to obtain the containers' confidential data both from persistent storage and RAM. The security of the platform itself, e.g, for C_0 and the CM, was already discussed in (Huber et al., 2015). According to 2, an attacker obtains physical access to the mobile device, implying that the adversary has the hardware, software, and thus all memory under control. We encrypt containers when the user actively switches them to background, locks the phone, or leaves it idle for a specific time. The attack starts either when the device with encrypted containers is left unattended or gets stolen. Containers share no sensitive data with other entities, because the memory usage of physical pages is tied to container boundaries. Since we encrypt all relevant memory segments (see 5), we fully cover the containers' sensitive data in RAM. F&C purges the unwrapped RAM encryption key in the kernel. Persistent memory is always encrypted due to FDE where the kernel stores the unwrapped FDE key. In our application scenario, the unwrapped FDE key in the kernel and the unwrapped RAM encryption key in user space are the only assets not protected through the encryption. However, at the moment the container encryption terminates, the CM purges these unwrapped keys, preventing the attacker from decrypting any persistent and volatile memory. The wrapped key complements can only be unwrapped using the SE. The attacker is possibly in possession of the SE, but lacks knowledge of the passphrase and cannot brute-force the SE. Hence, wrapped encryption keys are securely stored in main memory. This means that we need no special key storage, such as CPU registers. The CM only keeps the unwrapped key counterpart in RAM during encryption and when the container is unencrypted. Relocating the unwrapped key for that time, e.g., to the ARM TrustZone, does not increase the security, as sensitive data is in plaintext anyway.

Smartphones are suspended most of the time implying that the containers are already encrypted. When actively used, background containers remain encrypted and the foreground container gets either suspended by the active user or the system after a short idle-time. As evaluated in 7, the encryption process terminates quickly even for full containers. This means that the time frame to extract valuable data still in plaintext is impractical even when the attacker already starts the attack while the device is locking. Being in possession of the device, the attacker can hence only wait for incoming data, such as short messages or network traffic. Since sensitive data is generally end-to-end encrypted and encryption terminates inside the container, the attacker has no means to decipher that data. Unencrypted data, for example, short messages, can already be intercepted before it reaches the mobile device. In case of a violent offence, we assume that the victim was actively using a container. In this case, the background containers are protected while the actively used container remains unprotected if the system does not trigger the lock or if the victim does not manage to lock the device. The adversary

has no influence on processes of frozen containers, memory contents are not shared between containers and the key to decrypt pages of frozen containers is not present. A stolen device that was tampered with in the absence of its unsuspecting owner, for example, to intercept the key when the user returns, was not part of our threat model. We assured to leave no sensitive data behind by reading out process space as privileged user and by analyzing memory dumps of locked devices with the tool Volatility and with coldboot attacks, such as in (Huber et al., 2016). On common devices, we easily recovered lots of sensitive data, such as exchange passwords, FDE keys and further credentials. Even though in knowledge of the sensitive assets, we had no means to detect any sensitive data on devices protected by F&C.

9 CONCLUSION

We presented F&C, a novel mechanism for the encryption of sensitive data in main memory along with its successful application to protect mobiles device from physical attackers. F&C builds upon the freezer functionality of the Linux kernel making processes en- and decrypt their memory efficiently in parallel with a transient key. We synchronized the encrypting processes, ensured that frozen processes do not touch their memory and that external events, such as IPC, are deferred. The prototype we developed can be employed throughout different platforms, kernel versions and allows the selection of keys, processes and memory segments to be encrypted from user space. We extended an existing mobile device platform that runs multiple Android containers to integrate our prototype on smartphones. The platform allowed us to combine its virtualization and secure key management infrastructure with F&C in order to realize a fully functional system that thwarts physical attackers. We encrypted the containers that are not in active use and in order to maintain their full functionality, we inform the user with notifications about background events. In our security and performance evaluation, we showed that the encryption provides strong security for unattended devices and containers not in use. The average en- and decryption time of less than 2.5 seconds makes the prototype practical for daily use, especially in environments where the confidentiality of data plays a major role. We seek to integrate F&C into further scenarios on both embedded and desktop systems, for example into hypervisors to protect the full memory of different guest OSs.

REFERENCES

Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J. (2011). Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 173–187. ACM.

Apostolopoulos, D., Marinakis, G., Ntantogian, C., and Xenakis, C. (2013). Discovering Authentication Credentials in Volatile Memory of Android Mobile Devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*, volume 399 of *IFIP Advances in Information and Communication Technology*, pages 178–185. Springer.

Becher, M., Dornseif, M., and Klein, C. N. (2005). FireWire: All Your Memory Are Belong To Us. *Proceedings of CanSecWest*.

Boileau, A. (2006). Hit by a bus: Physical access attacks with Firewire. *Presentation, Ruxcon*.

Break & Enter (2012). Adventures with Daisy in Thunderbolt-DMA-Land: Hacking Macs through the Thunderbolt interface.

Chen, X., Dick, R. P., and Choudhary, A. (2008). Operating system controlled processor-memory bus encryption. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 1154–1159. IEEE.

Colp, P., Zhang, J., Gleeson, J., Suneja, S., de Lara, E., Raj, H., Saroiu, S., and Wolman, A. (2015). Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189. ACM.

Corner, M. D. and Noble, B. D. (2003). Protecting applications with transient authentication. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 57–70. ACM.

Devine, C. and Vissian, G. (2009). Compromission physique par le bus PCI. In *Procs. of SSTIC 09*. Thales Security Systems.

Götzfried, J., Dörr, N., Palutke, R., and Müller, T. (2016a). HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space. In *11th International Conference on Availability, Reliability and Security (ARES)*, pages 79–87.

Götzfried, J. and Müller, T. (2013). ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *Availability, Reliability and Security (ARES), 8th International Conf. on*, pages 161–168. IEEE.

Götzfried, J., Müller, T., Drescher, G., Nürnberger, S., and Backes, M. (2016b). RamCrypt: Kernel-based Address Space Encryption for User-mode Processes. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 919–924. ACM.

Gutmann, P. (1999). The Design of a Cryptographic Security Architecture. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99. USENIX.

Halderman, J. A., Schoen, S. D., Heninger, N., and et al, W. C. (2009). Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM. ACM*, pages 91–98.

Horsch, J., Huber, M., and Wessel, S. (2017). TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor. In *2017 IEEE Trustcom/BigDataSE/ISPA*.

Huber, M., Horsch, J., Velten, M., Weiß, M., and Wessel, S. (2015). A Secure Architecture for Operating System-Level Virtualization on Mobile Devices. In *11th International Conf. on Information Security and Cryptology - Inscrypt*. Springer.

Huber, M., Horsch, J., and Wessel, S. (2017). Protecting Suspended Devices from Memory Attacks. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, pages 10:1–10:6. ACM.

Huber, M., Taubmann, B., Wessel, S., Reiser, H. P., and Sigl, G. (2016). A Flexible Framework for Mobile Device Forensics Based on Cold Boot Attacks. *EURASIP J. Inf. Secur.*, pages 41:1–41:13.

Lie, D., Mitchell, J., Thekkath, C., and et al, M. H. (2003). Specifying and verifying hardware for tamper-resistant software. In *Security and Privacy. Proceedings. Symposium on*, pages 166–177. IEEE.

Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. (2000). Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 168–177. ACM.

Müller, T., Dewald, A., and Freiling, F. C. (2010). AESSE: A Cold-boot Resistant Implementation of AES. In *Proceedings of the 3rd European Workshop on System Security*, EUROSEC '10, pages 42–47. ACM.

Müller, T., Freiling, F. C., and Dewald, A. (2011). TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX.

Müller, T. and Spreitzenbarth, M. (2013). FROST: Forensic Recovery of Scrambled Telephones. In *Proceedings of the 11th Int. Conference on Applied Cryptography and Network Security*, pages 373–388. Springer.

Müller, T., Taubmann, B., and Freiling, F. C. (2012). TreVisor. In *Applied Cryptography and Network Security*, Lecture Notes in Computer Science, pages 66–83. Springer.

Ntantogian, C., Apostolopoulos, D., Marinakis, G., and Xenakis, C. (2014). Evaluating the privacy of Android mobile applications under forensic analysis. *Computers & Security. Elsevier*, pages 66 – 76.

Peterson, P. (2010). Cryptkeeper: Improving security with encrypted RAM. In *Technologies for Homeland Security (HST), IEEE International Conference on*, pages 120–126. IEEE.

Pettersson, T. (2007). Cryptographic Key Recovery from Linux Memory Dumps. Presentation, Chaos Communication Camp.

Simmons, P. (2011). Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 73–82. ACM.

Suh, E. G., O'Donnell, C. W., and Devadas, S. (2007). AEGIS: A single-chip secure processor. *Design & Test of Computers. IEEE*, 24(6):570–580.

Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., and Sarda, N. (2012). CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 77–91. USENIX.

Weinmann, R. (2012). Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*. USENIX.

Wessel, S., Huber, M., Stumpf, F., and Eckert, C. (2015). Improving mobile device security with operating system-level virtualization. *Computers & Security. Elsevier*.

Würstlein, A., Gernoth, M., Götzfried, J., and Müller, T. (2016). Exzess: Hardware-Based RAM Encryption Against Physical Memory Disclosure. In *Architecture of Computing Systems–ARCS*, pages 60–71. Springer.

Zhao, L. and Mannan, M. (2016). Hypnoguard: Protecting secrets across sleep-wake cycles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 945–957. ACM.