FridgeLock: Preventing Data Theft on Suspended Linux with Usable Memory Encryption

Fabian Franzen
Technical University of Munich
franzen@sec.in.tum.de

Manuel Andreas
Technical University of Munich
manuel.andreas@tum.de

Manuel Huber
Fraunhofer AISEC
manuel.huber@aisec.fraunhofer.de

ABSTRACT

To secure mobile devices, such as laptops and smartphones, against unauthorized physical data access, employing Full Disk Encryption (FDE) is a popular defense. This technique is effective if the device is always shut down when unattended. However, devices are often suspended instead of switched off. This leaves confidential data such as the FDE key, passphrases and user data in RAM which may be read out using cold boot, JTAG or DMA attacks. These attacks can be mitigated by encrypting the main memory during suspend. While this approach seems promising, it is not implemented on Windows or Linux.

We present FridgeLock to add memory encryption on suspend to Linux. Our implementation as a Linux Kernel Module (LKM) does not require an admin to recompile the kernel. Using Dynamic Kernel Module Support (DKMS) allows for easy and fast deployment on existing Linux systems, where the distribution provides a prepackaged kernel and kernel updates. We tested our module on a range of 4.19 to 5.3 kernels and experienced a low performance impact, sustaining the system's usability. We hope that our tool leads to a more detailed evaluation of memory encryption in real world usage scenarios.

ACM Reference Format:
Fabian Franzen, Manuel Andreas, and Manuel Huber. 2020. FridgeLock: Preventing Data Theft on Suspended Linux with Usable Memory Encryption. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20), March 16–18, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3374664.3375747

1 INTRODUCTION

Our society and businesses rely on the availability of secure computing devices such as notebooks or desktop PCs. As data theft may result in disclosure of important business secrets or sensitive personal information, these devices need to be protected from unauthorized access. A threat special to portable devices is that they can be easily stolen once unattended.

To counter the risk of data theft, many businesses and individuals rely on FDE to protect sensitive data on their devices. Without the appropriate passphrase or smart card, attackers will not be able to extract sensitive information from a switched off device. Depending on the Operating System (OS), FDE can be employed e.g. using Microsoft Bitlocker, Linux dm-crypt or Apple FileVault. Considering smartphones, both iOS and Android have also integrated FDE[8].

As a plus, iOS minimizes the presence of sensitive material in RAM using encryption aware storage controllers with secure RAM for key material. This reduces the attack surface to adversaries capable of obtaining a memory dump, but does not fully mitigate it as applications may still store sensitive information in RAM.

Common solutions do not protect this temporary data, such as passphrases or the FDE key, if the device is not fully switched off. We believe many users, not switching off their devices for faster wake up, are unaware of this attack surface. Attacks using physical properties of the hardware (e.g. cold-boot attacks [11]) or using unsecured peripheral connectors could be leveraged by an adversary to extract it. Such connectors could be JTAG, DMA-enabled ones (like Firewire [1] or Thunderbolt) or free memory DIMM sockets [22].

Unfortunately, hardware trust anchors like Trusted Platform Modules (TPMs) are usually only involved in the initial authentication process at boot. After successful authentication, the FDE key typically resides outside the TPM in normal unsecured RAM. IOS stores the FDE key solely on its Secure Enclave Processor[14], never exposing it to the main CPU. However, such special coprocessors are not available on off-the-shelf laptops or PCs.

Suspend-time memory encryption approaches, proposed in previous research, en-/decrypt memory during suspend and resume cycles. This has the advantage that once suspended, the FDE key no longer needs to be present on the system. As a result, even attackers with control over the CPU cannot read sensitive memory contents. Moreover, suspend-time encryption only impacts performance during suspension and resumption, but not during runtime. When pursuing the goal to protect unattended devices, suspend-time encryption represents an efficient and secure approach if specialized hardware is not available. Unfortunately, previous research only provided kernel patches for Linux, which have quickly become outdated as the kernel evolves. Therefore, we still lack an easy-to-use implementation to prove that this concept works in real-world setups. In summary, we make the following contributions:

- We provide FridgeLock, a tool to study the impact of suspend time memory encryption on real world setups.
- FridgeLock is designed as an LKM, such that suspend time memory encryption can easily be tested on a large number of Linux distributions without the need to recompile their kernel. We achieve this using DKMS to recompile the LKM in case of security updates. This results in a solution more agnostic to kernel changes.
- We tested our module on various distributions on the x86 platform and provide performance measurements, showing a user-acceptable performance for real world usage.

2 RELATED WORK

Several lines of work for protecting sensitive data in main memory exist. Key hiding techniques [7, 10, 17, 20] offload specific keys, such as the FDE key, from RAM to CPU or GPU registers and caches and execute the cipher exclusively on-chip. Besides being platform-dependent and hardly portable, key hiding mechanisms usually have a strong impact on performance. These mechanisms only protect a specific key and its associated cipher from memory attacks, but leave all other sensitive parts of main memory unprotected. Further, attackers gaining privileges on the system can directly access the keys, whereas with our approach, the memory encryption key is not available when the system is suspended.

Hardware-assisted memory encryption architectures [4, 21, 24] transparently encrypt main memory throughout the whole runtime of the system, keeping the encryption key and cipher computation off the main application processor. However, these architectures are not available on end user systems. Architectures designed by major hardware vendors, like AMD Secure Encrypted Virtualization (SEV) or Intel Multi-Key Total Memory Encryption (MKTME), target server systems only. Their goal is to protect memory against physical attackers and, at least in case of SEV, from a malicious or compromised hypervisor. SEV, for instance, stores the encryption key in an isolated coprocessor but has been demonstrated to be vulnerable against various side-channel attacks [15, 16, 23] allowing to extract encrypted memory in plaintext or to obtain privileges on encrypted guests. This shows that the reliable protection of main memory at all times poses a hard challenge.

Processors for consumer devices rather offer extensions to provide secure enclaves, such as ARM TrustZone, or Intel Software Guard Extensions (SGX). Enclaves enable shielded execution of sensitive code and the storage of data secure from both memory attacks and compromised OSs. These extensions can be leveraged as building blocks for security architectures, for instance, for use as isolated environments for encryption key storage and cipher execution [12].

Runtime memory encryption has also been realized in software. Some approaches leave a portion of memory unencrypted in a sliding window while most of the main memory is encrypted [6, 9, 18, 19]. In general, software-based runtime encryption approaches come with a notable impact on performance and can not provide memory protection against attackers gaining privileges on the system.

We base our memory encryption approach on *Freeze & Crypt* [13]. Further suspend-time approaches for x86 platforms are *Transient Authentication* [3] and *Hypnoguard* [25]. *Transient Authentication* encrypts main memory using a hardware token that provides the encryption keys [2]. When the token is removed, user space processes get suspended and their memory encrypted. Because suspension and resumption took about eight seconds, an application-aware mode was proposed. This allowed the protection of only specific assets using an API, which requires modifying applications.

Hypnoguard [25] hooks en-/decryption of memory into phases of suspension and resumption where the OS is no longer, respectively not yet, active. This allows to encrypt the whole memory without considering process mappings and without requiring kernel support but has the disadvantage that the kernel's support for hardware devices (such as displays, keyboards) is not available. The design requires implementing custom hardware-specific crypto routines and drivers to interact with hardware devices, such as for passphrase input. A TPM is used to protect the encryption while the cipher is executed in Intel's Trusted Execution Technology (TXT) environment. These design decisions make *Hypnoguard* less applicable in practice, while we require only adding an LKM to a system.

3 DESIGN

Like most of the previous approaches, FridgeLock targets Linux, because of the availability of source code. Further, we believe that memory encryption could be ported to other OSs if we can show feasibility on one of them. In this section, we derive the design of FridgeLock, based on the following design goals:

Easy Integration All existing academic approaches we are aware of are implemented as kernel patches, which forces the end user to recompile their kernel if they want to protect their system and, additionally, on every kernel update. To allow for a wide potential user base and to ease kernel updates with prepackaged distribution updates, we developed FridgeLock as an LKM. This allows distribution of FridgeLock as a binary or through dynamic compilation on every kernel update using DKMS.
Ultimately, we hope that FridgeLock will be integrated into the Linux kernel.

Low Performance Overhead We seek to keep the impact on performance as low as possible to not adversely impact user experience. Our LKM only hooks into the suspend and wakeup procedures, where small additional delays should be acceptable.

Protection of Sensitive Data Sensitive user data should be protected from adversaries under our assumed attacker model.

3.1 Attacker Model

We assume that an unattended device is stolen from a user in suspended state. In this state, the device is inspected by the attacker. Afterwards, the attacker can bypass all software and hardware mitigations (e.g. SEV) *somehow*. We only consider attacks where the device is lost *once* i.e. a device can not be stolen and given back. This excludes (1) *evil-maid attacks* where the attacker could e.g. corrupt the system and wait for the unknowing user to return, and (2) attacks involving an evil hardware vendor, who attacks the system from the HW side before it is stolen. As our design is not based on hardware trust anchors, we do not necessarily assume that TPMs or Secure Enclaves on the CPU are correctly implemented. An attacker may be able to execute arbitrary code (e.g. injected via JTAG or DMA) at any privilege level *after* the device is stolen.

Furthermore, our used cryptographic primitives (AES) and its operating mode (AES-XTS) have to be correctly implemented to be effective. This attacker model should be reasonable, given that e.g. TRESOR [17] was broken under similar assumptions [1].

3.2 Confidential Data in Memory

Given the goals and the attacker model, we evaluated the assets in the Linux kernel that need protection:

- **(A0) Filesystem.** The files on disk including sensitive user and system owned files. These files are protected by FDE.
- **(A1) Page Cache.** Opened files from disk may be cached in RAM in the page cache. As these are basically (partial) copies of files on disk, these can contain sensitive data.
- **(A2) Filesystem Metadata.** Metadata of the filesystem such as filenames, modification times and folder structure. Less critical than actual file contents, but nonetheless sensitive.
- **(A3) Userspace memory.** The active Virtual Memory Areas (VMAs) of running processes. May contain private keys, passphrases and (partial) copies of data from disk.
- **(A4) Free'd Pages.** The Linux kernel does not clear pages that have just been released by the kernel or by a userspace process (e.g. explicitly or process termination).
- **(A5) Kernel Objects.** For example, `task_struct`, i.e., internal information of running processes. It contains a CPU register snapshot, the process name, and a kernel stack reference.
- **(A6) Linux Keyring.** The Linux keyring is a central key storage inside the kernel. More specifically the CIFS filesystem uses the keyring to store passphrases to accessed shares.
- **(A7) Disk encryption keys.** The dm-crypt module is responsible for FDE and stores its keys outside of the Linux keyring.
- **(A8) Arbitrary Device Buffers.** Any hardware might store sensitive I/O-data such as keystrokes.

In contrast to these assets, e.g. the *Linux Text Segment* or the *Firmware and Bootloader Code* do not require protection under our chosen attacker model as we exclude *evil maid attacks.*

3.3 Integration into Linux Power Management

In order to put the system into the S3 sleep state (Suspend-To-Ram), the Linux kernel first suspends execution of userspace processes (called *freezing*). This is necessary to stop processes from interfering with the suspend process.[1] Kernel threads are not *frozen* by default, but *freezable* kernel threads (e.g. threads that could cause the suspend process to fail) are stopped directly after the processes. Finally, the kernel notifies device drivers to put their device into a power saving sleep state. When this process is finished, execution on the CPU is halted until an interrupt initiates the resume, where this procedure is done vice versa.

We split FridgeLock into two parts: An LKM and a non-encrypted *helper process* running in userspace. The LKM is responsible for process memory encryption, for protecting the other assets, and for spawning the helper process right before suspension. The helper process is responsible for memory and disk encryption key management during wakeup cycles. For this purpose, the userspace process queries the current user for the decryption passphrase after system wakeup.

The LKM integrates with Linux power management using the *device power management* subsystem (through `register_pm_notifier()`) and through the device driver power management API (i.e. `dev_pm_ops`). In order to get access to this API, we register a virtual device together with a driver FridgeLock provides. Combined, this offers the following hooking points crucial to our design:

[1]see `Documentation/power/freezing-of-tasks.txt` in the kernel sources for further information.

(1) On Early Suspend: Before the system is going to freeze the processes.
(2) On Late Suspend: After the system has frozen the processes.
(3) On Resume: Before the system is going to thaw the processes.

In the following, we describe FridgeLock's tasks from initialization to suspension and resumption:

Initialization Time. At its initialization, the LKM creates a character device for ioctl-based communication with the helper process to: (1) force the helper process to sleep until system resume, (2) to send the read-in FDE passphrase to our LKM after entry and (3) to probe for encrypted partitions needing protection. Moreover, it hooks into the *device mapper* infrastructure to obtain the FDE key on set, which we make use of to encrypt the userspace memory.

Hook 1. In case of a system suspend, the LKM is notified through hooking point (1) and starts the helper process. Moreover, it sets a bit on the helper process which causes the regular *freezer* to skip this process.

Hook 2. At hooking point (2), when all other userspace processes are *frozen*, we encrypt the memory map of the userspace processes (except for special mappings). Further, we evict filesystem caches and overwrite unused pages with zeros afterwards. As a last step, we wipe out the FDE key. For the actual encryption operations, we utilize the kernel crypto API. This allows using hardware crypto accelerators, such as AES-NI, resulting in extremely fast encryption speeds. We encrypt each page individually and utilize AES-XTS as cipher mode as our encryption requirements are almost identical to typical FDE, for which AES-XTS is recommended by NIST [5]. We use the physical page addresses as IVs to guarantee unique IVs for every page. To wipe and restore the FDE key, FridgeLock relies on the *suspend*, *resume* and *message* operations of the *dm-crypt* module.

Hook 3. On resume (3), the LKM wakes the helper process. The helper, in turn, asks the user for the FDE passphrase and restores the prior state of all dm-crypt devices using the regular *cryptsetup* toolchain. The LKM then decrypts userspace memory before the system returns to normal operation.

4 IMPLEMENTATION

In the following, we describe the implementation of our FridgeLock prototype, first our userspace helper followed by the LKM.

Userspace Helper. We partitioned our *userspace helper* into two parts, which we call *Stage 1* and *Stage 2.* Figure 1 reflects that the LKM spawns the *Stage 1* helper process at hooking point (1). The *Stage 1* process discovers the dm-crypt volumes on the system that need to be suspended, see Figure 1. Furthermore, it sets up an initramfs like tmpfs containing the *Stage 2* helper process and its necessary runtime environment, e.g. libcryptsetup. The *Stage 1* process then chroots into this new environment and executes the *Stage 2* process inside the chroot. This chroot operation is necessary because the rootfs is not available between wakeup and passphrase entry of the user. The *Stage 2* process is responsible for signaling the discovered volumes to the LKM via an ioctl, which will suspend and wipe the keys at hooking point (2). Further, the process asks the user for the passphrase for resumption (see hooking point 3,

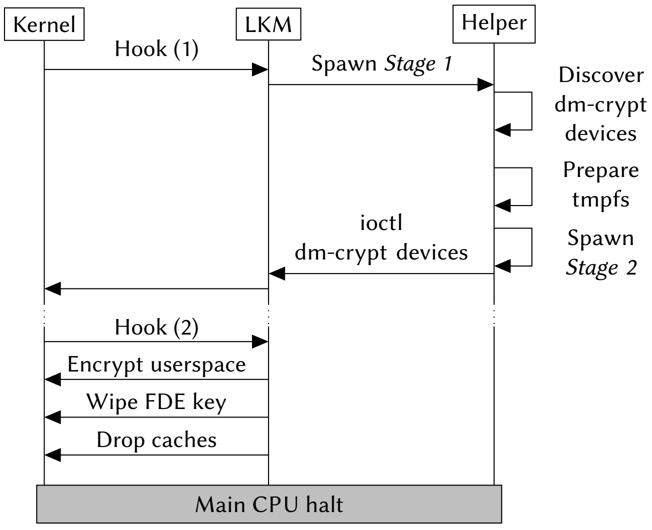

Figure 1: Sequence diagram of the suspend procedure.

omitted from Figure 1). On startup, the discovered volumes are received as parameters from the *Stage 1* process and transferred to the LKM via an ioctl. The LKM, in turn, suspends execution of the *Stage 2* process and returns control to the kernel.

Kernel Module. After our LKM gets notified of completed process freezing, see hooking point (2) in Figure 1, it iterates through the dm-crypt volumes received by the *Stage 1* process and suspends them using the device mapper kernel component. This suspend operation ensures that the encryption key is securely wiped from RAM and makes any further IO operation on the affected volume to be blocking until resumption. Before suspension of the *dm-crypt volumes*, it will place a kprobes based hook to extract the FDE key, which we use to encrypt userspace. The projects *arch-luks-suspend*[2] and its successor *go-luks-suspend*[3] provide an implementation to remove the FDE key during suspension. We utilized parts of their implementation to realize our *Stage 2* process.

Afterwards, the virtual address space encryption of all userspace processes takes place as follows. We start by iterating through all VMAs of our helper process. To avoid their encryption, we mark all pages belonging to those VMAs as "already encrypted" by setting a flag on the kernel `struct page`. In the next step, we iterate through all frozen userspace and kernel tasks, encrypt their VMAs and thus their pages in-place. Before carrying out the encryption of a page, we first check if the "already encrypted" flag is set, then if the page belongs to a special region (e.g. DMA or device memory), and if not encrypt it and set the flag. This approach ensures that neither the VMAs of our helper process nor mapped hardware memory, unaware of our encryption, are encrypted. On resume, we use our kprobes hook into the device mapper to observe the FDE key set of the helper process and re-obtain it in the LKM.

Certain functionality of our LKM requires usage of unexported kernel functions. In order to call these functions anyways, we utilized the exported kernel function `kallsyms_lookup_name`, which is able to resolve a symbol's name to its address in memory. We resolve all unexported functions at module initialization and are thus able to call them at any point through function pointers. If no specialized hooking mechanism is provided by the kernel, we use *kprobes* and *kretprobes* to intercept function calls.

We utilized the `/proc/sys/vm/drop_caches` mechanism to clear the page cache. Using this mechanism, the kernel can be advised to drop page cache, dentries and inodes from memory. We instrumented the `invalidate_page_cache` function, which is part of the inode clearing procedure of the kernel, using the kprobes framework. This instrumentation simply zeros out pages belonging to inodes that are going to be erased.

[2]https://github.com/vianney/arch-luks-suspend
[3]https://github.com/guns/go-luks-suspend

5 EVALUATION

5.1 Completeness of Protection

Our implementation addresses (A0), (A1), (A3), (A4), (A7) by construction. Asset (A2) is partially protected as dropping the page cache also contains inode and dentry structs. However, this does not include all filesystem metadata. Moreover, the Linux keyring (A6) and arbitrary hardware buffers (A8) are not sufficiently protected. In the case of (A6), an API to stop kernel threads from accessing the invalid keys that a key-wipe would leave behind is missing. In the case of (A8), various drivers store their buffers at arbitrary locations (e.g. in the device memory itself, the kernel heap or kernel stack). Protecting these buffers would require knowing every drivers exact implementation.

Furthermore, we experimentally verified our approach by comparing a QEMU snapshot of an un- and FridgeLock-protected virtual machine. In both cases, a test program loads known confidential data from disk and places them in memory. Moreover, several applications like Firefox are started. While a scan of the unprotected dump did reveal the FDE key and multiple copies of the confidential test data, the protected instance did not. Additionally, we analyzed the snapshots for secrets of the remaining applications using *AESKeyFinder* (proposed by [11]), which searches for expanded AES keys. Our results showed that the unprotected snapshot reveals several keys, while our protected snapshot does not.

5.2 Performance

We tested FridgeLock on a Dell XPS 15 9550 with an Intel i7 6700HQ (2.6 GHz, 4 cores), 16 GB RAM (DDR4 2133MHz) and a PM951 Samsung 512GB SSD. The performance of FridgeLock is linearly dependent on the memory usage of the running processes, i.e., on the amount of memory to en-/decrypt.

Therefore, we constructed three scenarios to test FridgeLock: ① A minimal scenario with only a basic set of processes (init + bash without xserver, userspace footprint 77MB), ② an average load scenario (Gnome + a few Firefox instances, userspace footprint 5,5GB) and ③ a high load scenario where all RAM is occupied by userspace. The results are visualized in Figure 2. On system resume, about the same time span is needed for decryption of the processes. The overall decryption time is dominated by passphrase entry; the cryptographic operations take the same time as for encryption. Additionally, no time is spent on clearing caches during resume.

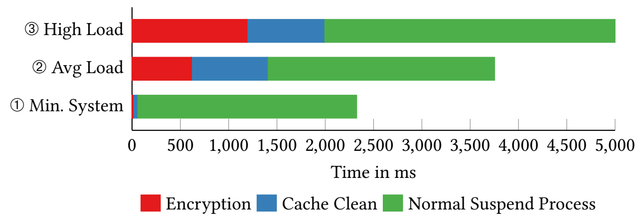

Figure 2: Overall measured system suspend time.

5.3 Maintainability

To implement our LKM, we were forced to load unexported functions via `kallsyms_lookup_name` and place `kprobes` hooks where necessary. These workarounds are generally discouraged as they are likely to break across different kernel releases, but are currently a necessity due to the lacking kernel API. While developing FridgeLock, we experienced both the kernel API and the unexported functions to undergo breaking changes, leading us to believe frequent maintenance of our LKM is necessary. We want to emphasize that this issue is just as present in a kernel patch implementation with the difference being that our LKM would possibly erroneously succeed building if an unexported function changed functionality or signature due to the nature of loading them via `kallsyms_lookup_name`. However, this could be avoided by additional automatic validation outside of compilation.

6 CONCLUSION

We presented FridgeLock, a tool for suspend time memory encryption on Linux. FridgeLock enables the protection of important assets in userspace and kernel memory on a suspended machine with deployed FDE. We successfully tested FridgeLock on x86 systems with kernel versions 4.19 to 5.3. Our evaluation on a mid-end notebook with 16GB RAM shows that FridgeLock's performance overhead is sufficiently small for use in practice, even in worst-case scenarios. This performance overhead is even more negligible considering it only affects suspend and resume operations.

Furthermore, our implementation as an LKM with userspace components results in an effortless installation and maintenance process for the end user through packaging and DKMS support. As usability is usually in direct conflict with security we deem the high usability of FridgeLock to be its strongest point.

Nonetheless, FridgeLock is currently not able to protect all sensitive assets in the kernel. First, the LKM design decision limits access to kernel internal structures. Even if the location in memory is known, we can not easily wipe information as we may not know all places where it is accessed in advance. Second, device drivers may contain numerous buffers with I/O data containing sensitive information through which we can not easily iterate. For instance, the keyboard driver may still contain the last typed passphrases before suspend. We consider the extension of the FridgeLock tool to locate and protect these buffers to be future work.

AVAILABILITY

To encourage open research, we open sourced our work at GitHub: https://github.com/fridgelock-lkm/fridgelock.

REFERENCES

[1] Erik-Oliver Blass and William Robertson. 2012. TRESOR-HUNT: attacking CPU-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 71–78.

[2] Mark D. Corner and Brian D. Noble. 2002. Zero-interaction Authentication. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking (MobiCom '02)*. ACM, 1–11.

[3] Mark D. Corner and Brian D. Noble. 2003. Protecting Applications with Transient Authentication. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys '03)*. ACM, 57–70.

[4] Guillaume Duc and Ronan Keryell. 2006. CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, 483–492.

[5] Morris J Dworkin. 2010. *Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices.* Technical Report.

[6] Johannes Götzfried, Nico Dörr, Ralph Palutke, and Tilo Müller. 2016. HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space. In *11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 79–87.

[7] Johannes Götzfried and Tilo Müller. 2013. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *Proceedings of the 2013 International Conference on Availability, Reliability and Security (ARES '13)*. IEEE Computer Society, Washington, DC, USA, 161–168. https://doi.org/10.1109/ARES.2013.23

[8] Johannes Götzfried and Tilo Müller. 2014. Analysing Android's Full Disk Encryption Feature. *JoWUA* 5, 1 (2014), 84–100.

[9] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. 2016. RamCrypt: Kernel-based Address Space Encryption for User-mode Processes. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, 919–924.

[10] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. 2014. Copker: Computing with Private Keys without RAM. In *NDSS*. 23–26.

[11] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.

[12] Julian Horsch, Manuel Huber, and Sascha Wessel. 2017. TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor. In *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '17)*. IEEE, 152–161.

[13] Manuel Huber, Julian Horsch, Junaid Ali, and Sascha Wessel. 2018. Freeze and Crypt: Linux Kernel Support for Main Memory Encryption. *Computers & Security* 86 (2018), 420 – 436. https://doi.org/10.1016/j.cose.2018.08.011

[14] Apple Inc. 2019. iOS Security - iOS 12.3. https://github.com/0xmachos/iOS-Security-Guides/blob/master/iOS_Security_Guide_May19.pdf

[15] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association.

[16] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec'18)*. ACM.

[17] Tilo Müller, Felix C Freiling, and Andreas Dewald. 2011. TRESOR Runs Encryption Securely Outside RAM.. In *USENIX Security Symposium*, Vol. 17.

[18] Panagiotis Papadopoulos, Giorgos Vasiliadis, Giorgos Christou, Evangelos Markatos, and Sotiris Ioannidis. 2017. No Sugar but All the Taste! Memory Encryption Without Architectural Support. In *Computer Security – ESORICS 2017*. 362–380.

[19] Peter A. H. Peterson. 2010. Cryptkeeper: Improving security with encrypted RAM. In *IEEE International Conference on Technologies for Homeland Security (HST)*. 120–126.

[20] Patrick Simmons. 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)*. ACM, 73–82.

[21] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. 2007. Aegis: A Single-Chip Secure Processor. In *IEEE Design & Test*, Vol. 24. IEEE Computer Society Press, 570–580.

[22] Anna Trikalinou and Dan Lake. 2017. Taking DMA attacks to the next level. *BlackHat USA* (2017).

[23] Jan Werner, Joshua Mason, and et al. 2019. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS 2019)*. 73–85.

[24] Alexander Würstlein, Michael Gernoth, Johannes Götzfried, and Tilo Müller. 2016. Exzess: Hardware-Based RAM Encryption Against Physical Memory Disclosure. In *Proceedings of the 29th International Conference on Architecture of Computing Systems (ARCS '16)*, Vol. 9637. 60–71.

[25] Lianying Zhao and Mohammad Mannan. 2016. Hypnoguard: Protecting Secrets Across Sleep-wake Cycles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 945–957.