Protecting Suspended Devices from Memory Attacks

Manuel Huber, Julian Horsch and Sascha Wessel
Fraunhofer AISEC
Garching near Munich, Germany
{firstname.lastname}@aisec.fraunhofer.de

ABSTRACT

Today's computing devices keep considerable amounts of sensitive data unencrypted in RAM. When stolen, lost or simply unattended, attackers are capable of accessing the data in RAM with ease. Valuable and possibly classified data falling into the wrongs hands can lead to severe consequences, for instance when disclosed or reused to log in to accounts or to make transactions. We present a lightweight and hardware-independent mechanism to protect confidential data on suspended Linux devices against physical attackers. Our mechanism rapidly encrypts the contents of RAM during suspension and thereby prevents attackers from retrieving confidential data from the device. Existing systems can easily be extended with our mechanism while fully preserving the usability for end users.

CCS Concepts

•**Security and privacy** → **Operating systems security;**

Keywords

Data Security, RAM Encryption, Systems Security, Data Confidentiality, Operating Systems Security

1. INTRODUCTION

The confidentiality of data on our today's computing devices not only depends on the widely used Full Disk Encryption (FDE), but also heavily relies on keeping the contents of volatile memory secret. The data temporarily stored in RAM is kept in plaintext and includes sensitive credentials, pictures, passwords, or key material [27, 29]. These valuable contents of RAM are an attractive target for memory attacks, such as via Joint Test Action Group (JTAG) [20], Coldboot [18, 8, 24], or Direct Memory Access (DMA) [1, 3, 15, 21]. Especially threatened are unattended or stolen laptops and desktop PCs in private, corporate or governmental use where private or classified information is stored and processed.

To mitigate this problem, we propose a lightweight kernel mechanism that safeguards unattended devices from memory attacks. This prevents attackers with physical access to the absent user's device from disclosing the secrets in RAM, which can cause severe harm. When the user or the idle system suspends a device, we encrypt the confidential data in RAM with the FDE key and remove it afterwards from RAM. We request a passphrase from the user to restore the key and initiate the decryption while waking up the system. The mechanism protects all userspace process memory, the valuable assets in kernel memory and removes other sensitive memory remnants, such as cipher states. Our primary design characteristics are the following:

Lightweightness. The mechanism needs no user configuration, leverages the existing OS infrastructure and can be easily added to systems in use.

Hardware Independence. The implementation is hardware independent and thereby ensures simple portability to other devices.

Usability. The user is not adversely affected in his workflow and is only required to type the FDE passphrase upon wakeup.

Performance. The mechanism sustains the almost seamless suspend/wakeup cycles and uses hardware accelerators for encryption and decryption, if present.

We implement our mechanism for Linux x86 devices, but its design can likewise be transferred to other OSs and platforms. Additionally, we propose a system based on Linux as a hypervisor in which we integrate our mechanism to protect the full memory space of guest OSs, such as Windows. In particular, our contributions are:

- The generic design of the lightweight RAM encryption mechanism.
- The implementation of a prototype for current Linux kernel versions.
- The integration of the mechanism into a hypervisor to solidly protect other OSs.
- A representative performance evaluation of the prototype along with a security discussion.

EuroSec'17, April 23 2017, Belgrade, Serbia

DOI: http://dx.doi.org/10.1145/3065913.3065914

The paper is organized as follows. We first focus on related work in Section 2 before describing the threat model in Section 3. Next, we present the design and implementation of our mechanism in Section 4 and Section 5. In Section 6, we present the virtualized environment in which we integrate our mechanism to protect guest OSs. We evaluate the performance and security of our prototype in Section 7 before concluding in Section 8.

2. RELATED WORK

Previous work on memory protection ranges from hardware-based to software-based mechanisms and varies in the scope of data under protection. We first discuss hardware-based protection mechanisms before discussing approaches implemented in software.

The main characteristic of hardware-based mechanisms, such as XOM [4] or Aegis [5], and many more approaches [7, 28, 2] is that sensitive data can be found unencrypted only in a single trusted component, namely the processor chip. These approaches protect all main memory. However, these architectures must be supported by hardware design, are difficult to realize and are mainly designed for special purposes, such as DRM protection. On common consumer devices, such as desktop PCs, smartphones or laptops, these mechanisms are usually not available.

Various processors for consumer devices are equipped with trusted extensions, such as the ARM TrustZone [22], or Intel SGX [6] which provide secure enclaves. These extensions make it possible to thwart memory attacks, but constitute only a building block for memory protection. On OSs which support the processor extensions, it is possible to move limited sensitive data of processes to an area in the enclave's hardware-protected memory. Developers must specifically design software for these enclaves, which are currently still limited, e.g., in their memory size. Additionally, these solutions all heavily depend on the platform and hardware they are based on.

Software-based mechanisms do not rely on these hardware features. An important aspect of previous work in this area is to examine the scope of data being protected. Some approaches focus exclusively on the protection of a specific encryption key stored in RAM, e.g., the FDE key, but leave all other data in main memory unprotected from memory attacks. Approaches for x86 [25, 12, 13, 23], as well as for ARM [11] and hypervisors exist [26]. Their common idea is to store a key in the CPU/GPU registers or in the CPU cache and to implement the cipher associated with the key on-chip at the cost of system performance.

The following approaches cover a broader scope of memory and encrypt the memory of running processes [19, 10, 17, 29]. However, Cryptkeeper [19] and Hypercrypt leave undefined portions of memory unprotected at all times and RamCrypt [10] only encrypts anonymous non-shared memory regions. Sentry [17] and CleanOS [29] are specifically tailored to the mobile domain. Sentry leverages special (legacy) hardware characteristics and CleanOS relies on cloud servers for key management and only covers parts of sensitive data. In addition to possibly leaving essential parts of memory unencrypted, all approaches notably and adversely affect runtime performance. In contrast, our mechanism covers all user space memory regions, sensitive data in kernel memory and does not impair runtime performance.

Transient Authentication [16] suspends and encrypts applications in the user's absence. However, the focus is on utilizing and recognizing a special hardware token for providing encryption keys. Despite that the OS is not completely suspended and that management tasks continue running, suspending/waking takes about 8 seconds. Therefore, an application-aware mode is proposed where developers must manually protect the assets of their applications.

Closest to our approach is Hypnoguard [14], which also en-/decrypts memory during the ACPI S3 suspension/wakeup. The mechanism hooks into this process at stages where the OS is not active. This means that there is no built-in support or driver for the interaction with hardware devices, such as (VGA/HDMI) displays or (USB/Bluetooth) keyboards. Therefore, their design requires to implement highly hardware-specific crypto routines for cryptographic hardware accelerators as well as numerous device drivers to interact with hardware devices, e.g., for passphrase input and to display the passphrase prompt. The encryption key is bound to a Trusted Platform Module (TPM) and encryption is executed in Intel's TXT environment. In contrast to our approach, this makes the approach highly hardware specific and particularly cumbersome for portability and when interacting with different hardware devices.

3. THREAT MODEL

Our attacker has physical access and starts his efforts on gaining confidential data when a suspended device, not previously tampered with, is unattended. Having sufficient time, the attacker can exploit software and hardware vulnerabilities, e.g., via JTAG [20], Coldboot [18, 8, 24], or DMA [1, 3, 15, 21], to gain access to both volatile and persistent memory. However, the attacker can neither break cryptographic primitives nor modify the memory to execute evil maid attacks waiting for the user to return (for the latter, we refer to system hardening techniques). A device once tampered with is hence no longer trusted, for instance after longer absence through theft, loss, or because the user notices the tampering attempt.

4. DESIGN

Today's Linux devices usually use FDE in combination with Linux Unified Key Setup (LUKS) to protect the confidentiality of persistent data. Our idea is to extend the coverage of FDE encryption of the data (and swap) partition to main memory contents when the device is suspended. When powering on, the system boots normally, i.e., the user is queried for the FDE passphrase after starting the initial ramdisk. By entering the valid passphrase, the system derives the FDE key and boots the OS. Figure 1 shows how we integrate our idea into the wake and suspend transitions of an existing system. While the device is actively used, the processes are not encrypted and the FDE key is present to transparently encrypt storage. After an idle time or active suspension, the common ACPI S3 suspend procedure starts. The kernel freezes its tasks, i.e., processes and threads, concurrently by sending every task a signal causing them to switch from user space execution to enter a non-schedulable state in kernel space.

At this point of state transition, we make each user space process encrypt its associated memory regions in its own

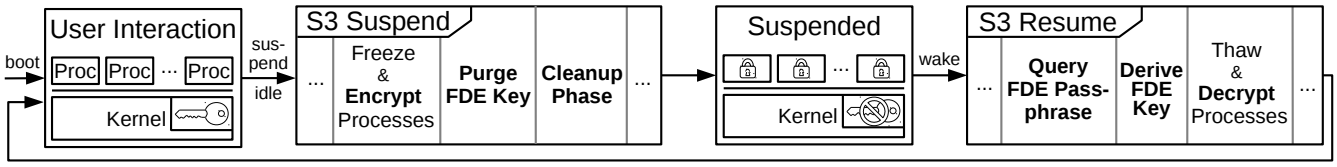

Figure 1: Overview on the steps (marked in bold face) introduced during suspension and resumption of a device.

context, i.e., anonymous, mapped, or shared memory and so on. Threads share the memory regions and kernel structures with their parent processes and siblings. To avoid multiple encryptions of the same regions, only the last task (of a process) to enter the *frozen* state encrypts the associated memory regions. A further challenge is that the physical pages to be encrypted can be shared across process boundaries. Therefore, every encrypting task marks its present pages as *encrypted*, if not marked before. Other tasks considering to encrypt a shared page skip already marked pages. As soon as every process has finished encryption, we purge the FDE key and further memory possibly containing sensitive data in a cleanup phase. After that, the common S3 suspension continues. All confidential data on the device is encrypted or removed.

Upon waking up the device, the S3 resumption is triggered. Before thawing (i.e., waking) the processes, we query the FDE passphrase from the user, as Figure 1 depicts. A secure routine takes the passphrase and derives the key with the Password-Based Key Derivation Function 2 (PBKDF2), which is used to decrypt LUKS headers. The decrypted LUKS header contains the FDE key, which we re-supply to storage encryption (`dm-crypt`) and use for the decryption of the processes while thawing (i.e., resuming) them. During decryption, each previously encrypting task decrypts the same set of pages and resets their flags to *decrypted*. After the decryption, the S3 wakeup continues and the system gets back to operation. The mandatory passphrase query allows to disable the screen lock after suspend requiring the end user to enter only a single passphrase.

We intentionally did not refer to a TPM or to other Secure Elements (SEs) for key protection to present a hardware-independent approach, especially for cases where a TPM is not required, available, or admitted. The security of the FDE key hence correlates with the complexity of its associated passphrase. However, a TPM can be easily combined with LUKS[1]. This implies that our proposed design can be easily adapted for cases where a TPM is desired and thus prevent brute-force attacks on the FDE key. In addition, the concept does not depend on processor extensions, such as Intel TXT or AMD SVM, to protect the system against the defined attacker.

5. IMPLEMENTATION

We implemented a prototype for recent Linux kernel versions (e.g, version 3.16 and 4.5, 32 and 64 bit). In this section, we focus on the three crucial steps of the implementation: The process *en-/decryption* procedure, the *cleaning* of further sensitive data and the *FDE passphrase query* for restoring the FDE key.

[1]https://github.com/shpedoikal/tpm-luks

Process Encryption and Decryption.

During suspension, every frozen task increments a counter we integrate into a structure shared with all tasks of a process. Every frozen task then compares the incremented counter with the number of tasks associated with the structure. In case the values equal, a task marks itself as the en-/decrypting task and starts the encryption of the Virtual Memory Areas (VMAs). The VMAs represent the user space memory regions, such as the stack, heap, code, shared and further anonymous or mapped memory segments.

The tasks responsible for en-/decryption iterate over their VMAs and asynchronously en-/decrypt the associated present pages using the kernel crypto API on all available cores. The only exception for excluded VMAs are non-confidential ones containing special segments and shared library code. The latter segments solely contain constant read-only data while special VMAs map memory shared with hardware devices (e.g., memory-mapped I/O or DMA memory). These VMAs can not be encrypted, because devices are not aware of the encryption and writing that memory likely corrupts the system. Encrypting the whole process memory as a single chunk would compromise these segments and fault on non-present pages.

The kernel considers a task frozen/thawed when all requests of a task are processed. We use AES in CTR mode as cipher and physical addresses as Initialization Vectors (IVs). Hence, we provide different IVs for all blocks to be encrypted. The crypto API selects the preferred cipher driver and available hardware accelerators, AES-NI instructions in our case.

Cleanup Phase.

After the encryption, sensitive data may still persist in regions not covered by the tasks' encryption. First, we zero out the FDE key. Second, we remove the remnants of the used ciphers: we zero the utilized cipher structures and also the relevant kernel stack regions using the kernel stack pointer. This makes it infeasible to deduct the encryption key from such information. Third, sensitive data may still exist in free pages, which were previously used by the kernel or user space processes and then freed. To remove these contents, we walk the list of free pages maintained by the page allocator and zero them out, which can be accomplished with very high throughput [17]. This step can be omitted when secure deallocation [9] is in place.

Passphrase Query.

We implement a simple passphrase query in the kernel comparable to [25] and derive the FDE key based on the supplied passphrase. The kernel and its drivers are fully operational at this stage and only user space processes are frozen. We easily reuse the drivers and software stacks in the

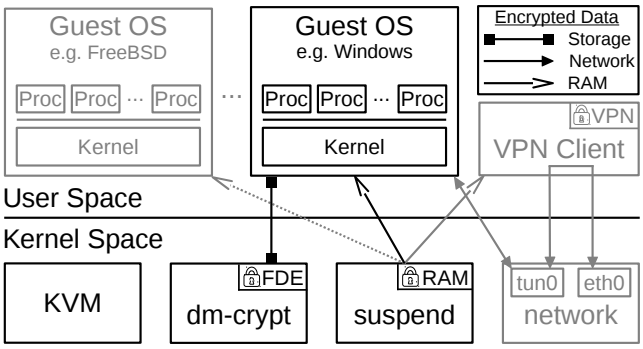

Figure 2: Application of our mechanism to solidly secure virtual guest OSs.

Measurement	Minimum	Maximum	Average
Processes	83	127	105
VMAs Encrypted	11,899	28,245	20,379
VMAs Total	15,241	35,050	25,548
Pages Encrypted	110,148	1,640,439	898,384
Suspend Time [ms]	142	1,497	807
Wakeup Time [ms]	125	1,373	745
Enc. Speed [MB/s]	2,860	5,106	4,453
Dec. Speed [MB/s]	3,081	5,387	4,824

Table 1: Minima, maxima and averages over all measurements on encryption performance, processes, VMAs and pages.

kernel to display the user a password prompt and to utilize connected keyboards. For using a sophisticated GUI and for re-plugging input devices, e.g., (bluetooth) keyboards, during suspension, it is possible to keep relevant processes (`udevd`, `bluez`) unencrypted and to thaw them in time when ensured that the daemons keep no confidential data. This also allows for using SEs, e.g., a smartcard, for two-factor authentication.

6. PROTECTION OF OTHER OPERATING SYSTEMS

Our mechanism can be easily leveraged to protect other OSs, such as Windows, running unmodified in a virtualized environment. Figure 2 shows the involved components where a minimal Linux serves as a hypervisor using Kernel-based Virtual Machine (KVM) for a guest OSs. The gray elements in Figure 2 constitute optional components, such as multiple guest OSs on the system. The Linux system only appears to the user when asking for the FDE passphrase on boot and wakeup. At any other time, end users work with their OS of choice with almost native performance. This makes it possible to, e.g., transparently protect Windows systems without impairing the usability for Windows users. When the device and hence the guest OS gets suspended, the full memory of the guest (including its kernel memory) is hence encrypted with our underlying mechanism. The FDE, i.e., `dm-crypt`, transparently encrypts storage and because our mechanism purges the FDE key after suspension, the guest OS's storage is fully secured from memory attacks. In addition, there is no confidential data stored in the Linux hypervisor itself.

To further protect the system, the optional components in Figure 2 emphasize that the guest's network traffic can be transparently protected by a user space Virtual Private Network (VPN) client, e.g., OpenVPN. The VPN client encrypts and routes the guest OS's network data over a secure VPN tunnel (e.g., through a corporate network). The Linux hypervisor automatically establishes the VPN connection such that common end-users do not need to provide further credentials and always benefit from the secure connection. Since the network credentials are part of the user space VPN process, the corresponding key is also encrypted during suspension.

7. EVALUATION

We first present our performance measurements before evaluating the security.

Performance Measurements.

We run a Debian userland with a Linux 4.5 kernel (4 KB page size) on a Lenovo T450s notebook (Intel Core i7-5600U CPU @ 2.60 GHz, dual core; 12 GB DDR3 RAM 1600 MHz). To fill and stress the RAM in a representative manner, we deployed a desktop environment and installed numerous applications, including business and graphics tools, browsers and virtual machines. Figure 3 shows the timing of 100 suspend and wakeup cycles of different sessions we measured using AES in CTR mode with a 256 bit key size. After booting the system, the suspension and wakeup of processes takes about 150 ms, where about 125,000 pages (500 MB) are encrypted. The trendlines show the linear increase in suspension and wakeup times when loading more and more applications and data into RAM. When the memory is under high load, i.e., 10-12 GB reserved, the suspension/wakeup took about 1.3 s. In such cases more than 1,600,000 pages (6,4 GB) are encrypted. The non-encrypted memory consists of non-confidential VMAs (special mappings and library code) and kernel memory. With our encryption model, we have fine-grained control over the regions requiring protection and do not need to encrypt the full space. Table 1 averages all measurements from Figure 3 and shows minimum and maximum values. In an average suspension cycle, 105 processes were suspended which encrypted about 900,000 pages. One cycle involved the encryption of more than 20,000 out of total 25,548 VMAs. The average suspend time was 807 ms and even faster 745 ms for the wakeup. The mechanism virtually reaches an en-/decryption speed of 4,453 MB/s, resp., 4,824 MB/s. Compared to the maximum of about 2.6 GB/s on one core (`tcrypt` benchmarks), our mechanism performs well and the system remained fully stable at all times. Ideas for future speed-ups are to zero out and unmap page-mapped files instead of encrypting them, and to decrypt pages on the fly after the wakeup. However, both approaches would affect runtime performance, because the mapped pages must be reloaded from persistent storage and in the latter case, pages are decrypted on demand.

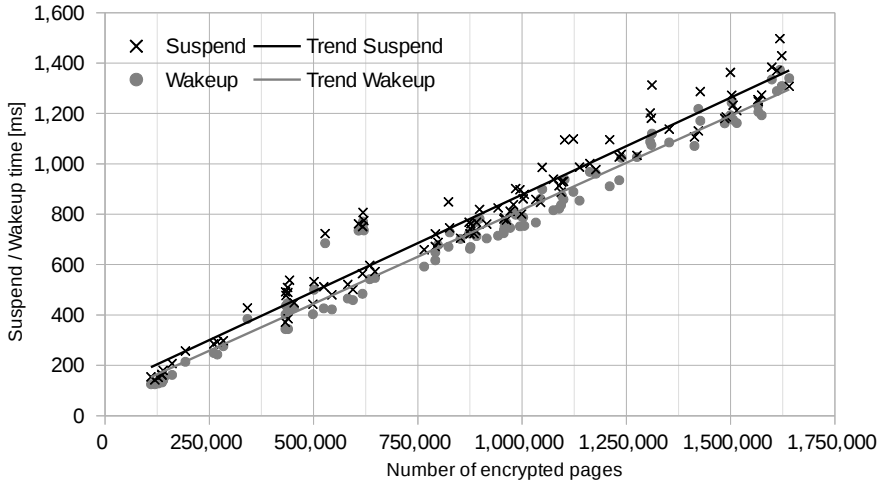

Figure 3: Suspend (crosses) and wakeup (circles) times in milliseconds in relation to the number of en-/decrypted pages.

Security Discussion.

With physical possession of the suspended device, our considered attacker (cf. Section 3) trying to obtain confidential data can possibly access persistent and volatile memory, categorized as follows:

Persistent Memory. It is protected by FDE and we removed the FDE key from kernel memory. The only way to decrypt storage is to decrypt the persistent LUKS header of the storage volume. This depends on the complexity of the FDE passphrase. Brute-force attacks take much effort, because the header decryption key is derived using the slow PBKDF2. A high number of iterations for PBKDF2 further slows down the repetition rate. In our design, we emphasized the possibility to easily integrate a TPM or to use an SE to prevent brute-force attacks.

Process Memory. All confidential VMAs were encrypted using AES in CTR mode with unique IVs. For the encryption, the FDE key was used and removed afterwards. The efforts hence coincide with the decryption of persistent memory.

Freed Memory. Since we zeroed out freed pages, there is no data remaining.

Kernel Memory. This part is not encrypted. However, we removed the FDE key, other possible key material, and remnants of our cipher operations.

In sum, the attacker not only has to possess the device for long time, but also invest remarkable effort to retrieve confidential data.

8. CONCLUSION

We presented an easily deployable mechanism that reliably protects confidential data on suspended devices against physical attackers. Based on the Linux kernel's FDE infrastructure, our kernel extension reliably protects the FDE key and encrypts the confidential data in volatile memory. With an average suspend and wakeup time of about 800 ms and rare peaks of less than 1,5 s on mid-performance devices with large RAM, our hardware-independent mechanism forms a fast, secure and portable solution. Existing Linux systems need no more than a kernel update. To wake up from suspend, the end user solely provides the FDE passphrase. This not only preserves high usability of the system, but also prevents end users disabling the OS screen lock from opening doors for everyone after suspension. We emphasized the possibility to combine our mechanism with a TPM or SE to prevent brute-force attacks on the FDE key derivation. For end-users of, e.g., Windows systems, we proposed a transparent, secure and lightweight virtualization solution where the full memory of guest OSs likewise benefits from the RAM encryption mechanism.

9. REFERENCES

[1] A. Boileau. Hit by a bus: Physical access attacks with Firewire. *Presentation, Ruxcon*, 2006.

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

[3] C. Devine and G. Vissian. Compromission physique par le bus PCI. In *Procs. of SSTIC '09*. Thales Security Systems, 2009.

[4] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz. 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, 2000.

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

[6] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd*

Int. Workshop on Hardware and Architectural Support for Security and Privacy. ACM, 2013.

[7] G. Duc and R. Keryell. CryptoPage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In *Computer Security Applications Conference. ACSAC'06. 22nd Annual*, pages 483–492. IEEE, 2006.

[8] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino et al. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM. ACM*, pages 91–98, 2009.

[9] J. Chow, B. Pfaff, T. Garfinkel and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 22–22. USENIX Association, 2005.

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

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

[12] L. Guan, J. Lin, B. Luo and J. Jing. Copker: Computing with Private Keys without RAM. 2014.

[13] L. Guan, J. Lin, B. Luo, J. Jing and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *2015 IEEE Symposium on Security and Privacy*, pages 3–19. IEEE, 2015.

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

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

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

[17] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189. ACM, 2015.

[18] P. Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, page 4. USENIX Association, 2001.

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

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

[21] P. Stewin and I. Bystrov. Understanding DMA malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012.

[22] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security – Enabling Trusted Computing in Embedded Systems. 2004.

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

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

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

[26] T. Müller, B. Taubmann and F. C. Freiling. Trevisor. In *Applied Cryptography and Network Security*, Lecture Notes in Computer Science, pages 66–83. Springer, 2012.

[27] T. Pettersson. Cryptographic Key Recovery from Linux Memory Dumps. Presentation, Chaos Communication Camp, August 2007.

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

[29] Y. Tang, P. Ames, S. Bhamidipati et al. 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, 2012.