RamCrypt: Kernel-based Address Space Encryption for User-mode Processes

Johannes Götzfried
FAU Erlangen-Nuremberg
johannes.goetzfried@cs.fau.de

Tilo Müller
FAU Erlangen-Nuremberg
tilo.mueller@cs.fau.de

Gabor Drescher
FAU Erlangen-Nuremberg
gabor.drescher@cs.fau.de

Stefan Nürnberger
CISPA, Saarland University
nuernberger@cs.uni-saarland.de

Michael Backes
MPI-SWS
backes@mpi-sws.org

ABSTRACT

We present RamCrypt, a solution that allows unmodified Linux processes to transparently work on encrypted data. RamCrypt can be deployed and enabled on a per-process basis without recompiling user-mode applications. In every enabled process, data is only stored in cleartext for the moment it is processed, and otherwise stays encrypted in RAM. In particular, the required encryption keys do not reside in RAM, but are stored in CPU registers only. Hence, RamCrypt effectively thwarts memory disclosure attacks, which grant unauthorized access to process memory, as well as physical attacks such as cold boot and DMA attacks. In its default configuration, RamCrypt exposes only up to 4 memory pages in cleartext at the same time. For the nginx web server serving encrypted HTTPS pages under heavy load, the necessary TLS secret key is hidden for 97% of its time.

Keywords

RAM Encryption, Memory Disclosure Attacks, Data Protection, Data Lifetime, Physical Attacks

1. INTRODUCTION

Process isolation and access control have proven to be conceptually elegant and widely deployed principles for preventing one process from accessing another process' memory [19]. In practice, however, the improper deployment of access control and side effects of memory optimizations and frequently debugging undermine the principle of isolation, leading to unexpected disclosure of otherwise isolated memory [6]. Prominent examples of such inadvertent memory disclosures rely on established operating system design principles such as swap files and crash reports (so-called *core dumps*) that intentionally write process contents to disk, and thereby disclose process memory in plain. Once a swap file or core dump file exists on disk, it is only protected by logical means against illegal access; hence, it is susceptible to improper configuration of access control or to starting a secondary OS that is not constrained to adhere to the original access control restrictions. Similarly, kernel drivers, including those provided by third parties, are prone to error and can give attackers full access to the physical memory space. For example, Samsung's official firmware for the Exynos chipset used in Android phones of the Galaxy series disclosed memory to attackers by accidentally offering an unprotected `/dev/mem` device [1].

In addition to software-based attacks, an attacker can read the memory contents of the physical RAM, knowing that sensitive data of a process resides there at any given time. Such attacks range from exploitable Firewire devices that have direct memory access [3, 5, 28] to *cold boot attacks* which physically transplant RAM modules into another machine [14, 27, 13]. In either case, an attacker can completely recover arbitrary kernel and process memory.

Moreover, since RAM constitutes a shared resource and, hence, portions of RAM are frequently re-used, improper de-classification or insufficient deletion may cause the inadvertent disclosure of sensitive information [26]. To reduce the *data lifetime* of sensitive information used only briefly inside a process, traditional approaches that wipe this data are conceptually hard to implement in a transparent manner [18]. With transparent data encryption, however, all information a process contains is always encrypted, and automatically decrypted when accessed. With this approach, sensitive information that is used seldom can be safely hidden without requiring lifetime knowledge about it.

1.1 Our Contribution

In this paper, we propose RamCrypt, a kernel-assisted encryption of the entire address space for user-mode processes. In detail, our contributions are:

- Sensitive data within the address space of a process is encrypted on a per-page basis inside the OS page fault handler. Sensitive data within core dumps and memory that get accessed by debuggers, for example, are therefore protected.
- Only a small working set of pages, also called the *sliding window*, remains unencrypted at any given time. By

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

ISBN 978-1-4503-4233-9/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897924

default, the size of this sliding window is four pages.

- The prototype implementation of RamCrypt leads to a performance drawback of 25% for single-threaded tests of the *sysbench* benchmark suite. For multi-threaded tests, the performance drawback is generally higher and heavily depends on the chosen sliding window size. For non-protected programs, RamCrypt imposes no performance overhead.
- By using *CPU-bound encryption*, cryptographic keys are never stored in RAM, but solely inside CPU registers over the entire uptime of a system. All user-mode processes are protected with the same key, which is only visible in kernel mode.
- RamCrypt can be enabled on a per-application basis by setting a flag inside the ELF program header, without the need for binary rewriting or recompilation.

RamCrypt is free software that is published under the GPL v2 and is available at `https://www1.cs.fau.de/ramcrypt`.

1.2 Related Work

CPU-bound encryption is somewhat related to our work but only protects a small fraction of sensitive data and not, for example, a whole process address space. Symmetric CPU-bound encryption schemes rage from register-based schemes as operating system patch [21, 11] to hypervisor-based solutions [22, 10] and cache-based schemes [17]. There are even CPU-bound encryption schemes for asymmetric encryption algorithms such as CPU-bound RSA implementations that either are register based [9] or based on hardware transactional memory [12]. All these solutions, however, just keep the encryption key and intermediate data out of memory but no other sensitive information because the secure storage area these solutions make use of is limited.

Full memory encryption solutions are most related to our work. There are theoretical approaches [7] but also practical implementations to encrypt, for example, swap space [25]. Furthermore, a solution for embedded hardware [16] exists. The only solution, however, that also targets the Linux kernel [24] is not publicly available and does not store the encryption key outside RAM. Thus, it does not protect effectively against memory disclosure attacks.

We are fully aware of Intel's recently announced Software Guard Extensions (SGX) [20] which amongst others also encrypt the memory of software running within so-called enclaves. Currently, however, enclaves are restricted to a static size and cannot dynamically grow after they have been initialized. Furthermore, existing programs cannot simply be run within enclaves without modifications to either the program itself or at least the libraries, because the enclave environment is restricted and syscalls are forbidden. Consequently, Intel's SGX cannot provide binary compatibility which is one of the design principles behind RamCrypt.

2. BACKGROUND

This sections describes the necessary building blocks on which RamCrypt is based on. Readers familiar with *CPU-bound encryption* (Sect. 2.1) and *virtual memory management in Linux* (Sect. 2.2) may safely skip this section.

2.1 CPU-bound Encryption

We use CPU-bound encryption/decryption to prevent the cryptographic key and any intermediate state from ever being stored in RAM [21, 4, 23, 11]. In particular, RamCrypt is built upon the Linux kernel patch TRESOR [21], which is an implementation of the AES algorithm for the Linux crypto API. It only uses CPU registers to store the encryption key as well as any intermediate information like the AES key schedule. In fact, no cache or RAM is used to store any part of the key or any intermediate state of the AES computation. To prevent intermediate states of AES entering RAM due to context switching, TRESOR is executed inside an atomic section in kernel mode. Inside those atomic section, interrupts are disabled. According to the authors of TRESOR, however, the *interbench* test suite has shown that interrupts are disabled only briefly such that system interactivity and reaction times are not affected.

To bootstrap TRESOR, the user has to type a password during system start-up and after wake-up from suspend mode, which is then used to derive the key stored in the debug registers. We patched TRESOR in a way that also a random key can be generated during start-up. This key is protected against physical attacks like cold boot [14] because debug registers are cleared after a CPU reset. An attacker would require logical access to a machine and must be able to execute kernel code in order to read out the key.

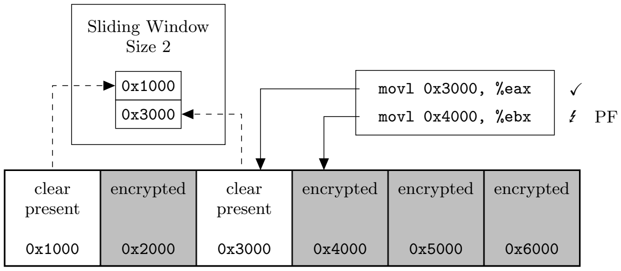
2.2 Linux Virtual Memory Management

On a modern CPU, process isolation is realized by simulating pristine address spaces to each process, which cannot interfere with each other. Every process starts with an empty address space that is divided into so-called *pages*. While executing code and accessing data, the necessary pages are loaded into the address space on demand by catching the first attempt to access them. Contiguous virtual pages do not necessarily have to be backed by contiguous physical pages. Each access to a page is first handled in hardware by the *Memory Management Unit* (MMU), which translates access to a virtual page into an actual physical page. Whenever such a translation fails, the OS page fault handler is invoked and determines whether missing pages need to be loaded or if an access to invalid memory occurred. Since every access to code and data has to go through the page fault handler at least once, this is a suitable part to incorporate RamCrypt. RamCrypt modifies the page fault handler such that not only first accesses are caught, but it additionally ensures to catch the next access to that page again by manipulating flags within the corresponding page table entry.

An important concept of Linux' memory management is *Copy-on-Write* (COW): A new process under Linux is created with the help of the `fork()` system call, which might be followed by a call to `execve()` to execute a new binary. To make `fork()` an efficient system call, the entire address space is not copied but only its virtual mappings point to the same physical pages. In order to avoid interference with the parent process, the cloned mappings are marked read-only, regardless of their original access rights. This ensures that read access can occur normally, while write attempts will be caught resulting in a transparent writable copy of the affected page.

3. DESIGN AND IMPLEMENTATION

In this Section, we describe the design and implementation of our RamCrypt Linux kernel patch. RamCrypt encrypts physical pages of a given process during its runtime and automatically only decrypts exactly those pages, which are currently accessed by the process. To this end, we leverage



```
movl 0x3000, %eax
movl 0x4000, %ebx
```

Figure 1: Sliding window. Next access will cause a page fault (PF) for page $0x4000$.

the existing Linux page fault mechanism, which already provides logic to transparently map not-yet-existing pages gradually into the address space. We augmented the binary *present/non-present* understanding of the kernel by another dimension: *encrypted/cleartext* data. This allows us to build upon the existing techniques that trap code, which tries to access non-present data – but with the additional distinction that this data might already exist, just in encrypted form. If so, it is automatically decrypted by the kernel such that access to that data seems to be in clear to the program.

RamCrypt can be activated for each process separately and, if activated, encrypts all pages associated with anonymous private memory. Anonymous private memory is the opposite of shared process memory in Linux. In other words, private memory is anything created by a process and includes the BSS segment, heap and other allocated memory, the stack, and private data of all loaded shared libraries. Shared memory mappings between processes and mappings that are backed by a file (e.g. code) are intentionally not encrypted by RamCrypt as these can intuitively be considered *'public'* anyway. In most practical scenarios, binary code is not worthwhile to be protected against memory disclosure attacks because virtually all programs in use today are widely available. Consequently, all sections of a program that can contain sensitive information are protected by RamCrypt and only code sections mapped as executable are left unmodified.

3.1 Sliding Window

The memory pages that have been encrypted by RamCrypt need to be decrypted when data stored inside them is about to be accessed. Due to limitations of the x86 architecture, one can only detect memory access on a page (4 kB) granularity. Therefore, RamCrypt decrypts an entire page whenever a process tries to access data stored within a still encrypted page. As long as data residing within that page is accessed, no measures need to be taken by RamCrypt since data is temporarily available in clear inside that page. As soon as data residing in another page is accessed, RamCrypt can decrypt that other page after encrypting the last accessed page again. This way, only one page at a time is available in clear. However, this strict encryption/decryption pattern is not practical for workloads that heavily access data residing in two or more different pages. To overcome this performance bottleneck, we introduce a *sliding window* per process (i.e., per virtual address space), which represents the last n pages that were accessed and are hence kept in clear. Every other page is always only kept in encrypted form. RamCrypt always ensures that accessing an encrypted page triggers a page fault and the sliding window is calculated anew. The size of the sliding window, which is illustrated in Figure 1, can be configured as a kernel boot parameter and is a trade-off between performance and security.

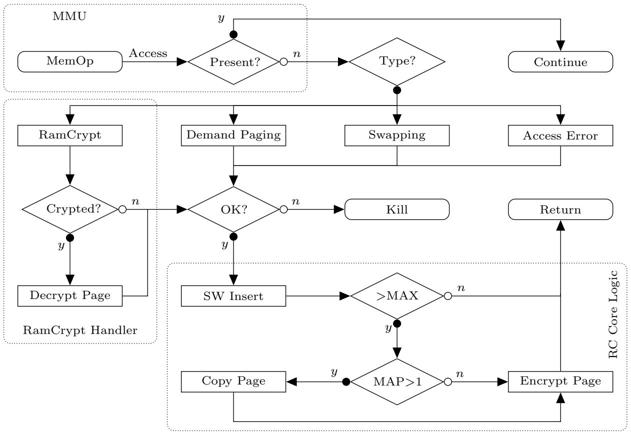


Figure 2: Flow diagram of how the MMU and the Linux kernel with RamCrypt interact.

3.2 RamCrypt Workflow

From a high level perspective, RamCrypt consists of two parts: (1) a modified page fault handler that handles accesses to encrypted pages, and (2) the RamCrypt core logic. Basically, the RamCrypt page fault handler decrypts the page for which a page fault occurred while the RamCrypt core logic determines which page needs to be encrypted again in order not to exceed the maximum number of n cleartext pages at a time. In detail, the new page fault handler decrypts a page and then makes it accessible by changing its corresponding page table entry. The RamCrypt core logic in turn checks whether the executed page fault handler has succeeded, then appends the new page to the sliding window and checks if and which page to remove from the window. Removal includes encrypting the page and clearing the *present* flag to ensure a page fault will be triggered the next time the page is accessed.

Note, that due to Linux' demand paging, it is sufficient to hook RamCrypt into the page fault handler as access to all new mappings always triggers a page fault. This also registers them in the sliding window mechanism. The overall process of how RamCrypt interacts with the remaining parts of the Linux kernel down to the hardware is shown in Figure 2.

3.3 Topping off RamCrypt

In order to address all eventualities of a program, the whole lifecycle of its data must be considered. This foremost includes the controlled deletion of data. To this end, RamCrypt scrubs de-allocated pages with zeros. This happens either if a memory region is deliberately freed by the programmer, or when a process is terminated. This security measure ensures no clear pages are leaked once memory pages became inaccessible to the program. Additionally, it makes decryption impossible should the key ever be retrieved.

While the overall concept of RamCrypt may sound simple, a lot of challenges had to be addressed in order to handle the inner peculiarities of memory management in a modern operating system. In particular, copy-on-write (COW), multithreading and multiprocessing make the abstracted sliding window approach that we presented so far more complex in reality. To meet the goal of backward compatibility with unmodified binaries, a lot of corner cases had to be added to the RamCrypt logic to support multiple threads within a single address space and all flavors of forking, which involves the creation of a new address space with temporarily shared COW pages. A naive implementation would destroy common COW semantics for newly `fork`'ed processes, as child and parent process share the same physical memory pages. Encryption in one process would lead to encryption in another process, of which the other process must be aware. To address this unwanted behavior, the COW semantics had to be made aware of the additional encryption dimension to copy data on encryption. Another challenge that arises is that `fork`'ed and `execve`'d processes might request RamCrypt encryption, while their parent was unencrypted. Here, COW cannot be used at all. Our implementation fully supports transitions from encrypted processes to non-encrypted processes and vice versa.

We use a variant of the TRESOR cipher (see Sect. 2.1) which is configured to operate like AES-128 in the XEX mode of operation. The initialization vector (IV), which is fed into the encryption routine to get the tweak for XEX, is the virtual address of the page that is about to be encrypted, concatenated with the PID of the currently running process. The reason we use virtual addresses instead of physical addresses for the IVs is that physical addresses may change over time when the kernel relocates pages. As a consequence of using virtual addresses, a single page cannot be mapped to two different virtual addresses within the same address space (because otherwise the decryption of both mappings would lead to different plaintext data). As shared memory regions are out of scope for our implementation, and since temporarily shared pages due to COW semantics are always mapped to the same virtual location, using virtual addresses as initialization vectors does not limit RamCrypt.

Using PIDs as part of the IVs prevents attackers, who have access to a physical memory disclosure vulnerability, to guess the content of an encrypted page by creating a malicious process that maps pages to the same virtual addresses. After reading out an encrypted page, both ciphertexts could be compared to conclude whether the plaintext data has been guessed correctly.

If a process forks, the child process uses the PID of the parent process instead of its own PID as part of the IV such that temporarily shared pages between the parent and the child process can be decrypted. When the child process executes another RamCrypt-enabled binary by calling `execve`, the child's own PID is used as part of the IV as no temporarily shared mappings between the parent and the child process are used anymore. The combination of a virtual address and the PID as IV ensures that no patterns of encrypted data can be extracted when using RamCrypt.

4. EVALUATION

In this Section, we evaluate RamCrypt regarding runtime performance and security (Sect. 4.2). All evaluations have been performed on a standard desktop computer with an Intel Quadcore CPU (Intel Core i5-2400) running at 3.1GHz and eight gigabytes of RAM. From the software side, we used an unmodified installation of Debian Wheezy with a base system installed and a RamCrypt patched kernel.

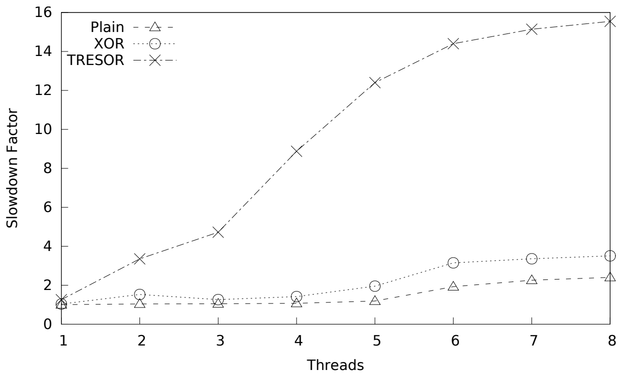


Figure 3: RamCrypt performance with sliding window size 4 and different encryption methods.

4.1 Runtime Performance

To evaluate the runtime performance of RamCrypt, we used the *sysbench* benchmarking utility that is shipped with most Linux distributions. The performance overhead is calculated as the relative run-time difference of the RamCrypt-enabled version of the benchmark compared to the unmodified version that ships with Debian. The unmodified version serves as ground truth to which the flagged executable is compared. To this end, a flag inside the ELF program header is set by means of a provided script.

Cipher Performance Impact.

In Figure 3, different encryption methods for RamCrypt are evaluated with the default sliding window size of four to show how big the overhead of the TRESOR cipher alone is and the slowdown factor compared to the ground truth is depicted. To this end, we ran RamCrypt with the TRESOR cipher enabled, with the identity function that outputs its input and with XOR encryption. The "encryption" using the identity function basically measures the overhead of the paging and RamCrypt logic but has almost no overhead due to the lack of CPU-bound encryption. Hence the name in the graph is 'plain' encryption. All measurements with RamCrypt enabled have in common that the sliding window size of four becomes a bottleneck for more than four threads as their concurrent execution competes for the next available window. The RamCrypt implementation that uses CPU-bound TRESOR encryption shows a slowdown of approximately 25% for the singlethreaded test. As expected, when the number of concurrent threads increases, the slowdown is larger due to the fixed-size sliding window. Note that RamCrypt is always enabled for the whole process, meaning for the main executable and all loaded libraries, and thus includes data of libraries, as well. The overhead would be significantly lower if only the main executable were protected. So this number constitutes a worst case scenario.

Sliding Window Performance Impact.

In figure 4, RamCrypt is evaluated with different sliding window sizes. For a sliding window size of 16, i.e., sixteen clear pages are available for up to eight threads, our implementation scales almost with the distribution version of sysbench. With eight threads, a slowdown of approximately 12% is reached. However, in a multithreaded process, the

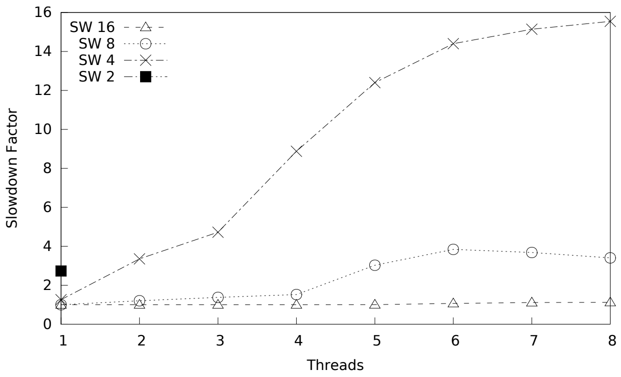


Figure 4: RamCrypt performance with TRESOR cipher and different SW sizes.

threads compete for the same amount of decrypted and readily accessible pages at the same time. For a sliding window size of two, sysbench is not able to launch more than one thread because it requires an additional control thread as well. In the singlethreaded run with a sliding window size of two, the slowdown factor is as high as 170%, while size four of the sliding window pushes the slowdown to acceptable 25%. With sizes of eight and sixteen, there was no slowdown measurable.

4.2 Practical Security Analysis

Due to the nature of RamCrypt, pages that are currently listed within the sliding window have to reside unencrypted in the RAM and thus might leak sensitive data over a limited time span. We therefore measured the relative time of unencrypted pages residing in memory for a real world application and specifically evaluated how long secrets are exposed in clear. We flagged the *nginx* web server to use RamCrypt and measured how long each page is accessible in clear, i.e., how long it was listed within the sliding window. To make nginx contain actual secrets, we set it up such that it delivers SSL-encrypted HTML pages under maximum load. The HTML document we transferred was a standard welcome site of nginx with a size of 151 bytes. Furthermore, we identified the memory pages in which the private exponent of the RSA key resides. We then observed how long these pages have been exposed to RAM in clear.

Table 1 shows the minimum, average and maximum relative time as part of the overall runtime of a page being exposed in clear for different sliding window sizes. Since nginx was used under maximum load, these numbers represent worst case exposure times for the key material, which under less load is consequently exposed shorter. From the minimum, maximum and standard deviation, we can conclude that some pages are used over the entire lifetime of a process while others are used only rarely, for example, during startup. The first row of table 1 shows how long the pages containing secret key material are exposed in clear over a process' lifetime. With increasing sliding window size, the average time of pages accessible in clear increases because more pages are stored in clear in parallel. For our default sliding window size of four, however, we consider a relative time frame of only 3% (where the secret is being exposed in clear) a good result. Although RamCrypt is not able to entirely prevent accessing the secret with the help of memory disclosure attacks, the bar for attackers is raised since precise timing becomes necessary. Furthermore, attacks like cold boot might entirely be prevented if the remaining 3% of cleartext data only resides in CPU caches but never enters RAM.

		Temporal Exposure per Page (%)		
		n=4	n=8	n=16
Secret Key Pages		3.07	14.37	21.68
All Pages	Min	0.0000	0.0005	0.0017
	Avg	7.63	12.66	17.95
	Max	99.83	99.76	99.99
	StdDev	19.77	21.82	25.43

Table 1: Temporal exposure of unencrypted pages (in %) for different sliding window sizes n. Measured *nginx* process served 1500 SSL requests.

5. DISCUSSION

Although RamCrypt is able to encrypt the data of whole process address spaces based on the user's choice, process-related data outside the virtual address space of a given process might still be accessible in clear. In particular, kernel or driver buffers and the buffers of peripheral components might contain sensitive data which cannot be protected by RamCrypt. Recently, it has been shown that the frame buffers of the VRAM can be recovered even after a reboot [2]. One approach to at least partially mitigate the issue of sensitive data stored in buffers outside the process' virtual memory is the use of ephemeral channels [8]. The downside, however, is that this approach requires modified peripheral devices and explicit use in source code.

Attacks such as the popular OpenSSL Heartbleed vulnerability (CVE-2014-0160) or other memory disclosure attacks [15] that are able to read out RAM with privileges of the attacked process cannot be prevented as RamCrypt transparently decrypts RAM when accessed by the owning process. There is no direct fix to this limitation as RamCrypt needs to act transparently to any process that is RamCrypt-enabled. To address this issue, source code needs to be changed, e.g., to protect itself against inadvertent access when the data is not needed by using CPU-bound encryption schemes [12, 9] that do not store keys in the address space of a process.

Although RamCrypt with a sliding window size of four imposes a performance overhead of only 25%, performance must still be improved for multithreaded applications. As shown in Sect. 4.1, the performance is strongly influenced by the cipher that is used. Consequently, speeding up the TRESOR cipher would also give a performance boost to RamCrypt. To speed up TRESOR, hardware transactional memory could be exploited as it has been done for a CPU-bound RSA implementation [12]. Another possibility is to compute the key schedule for a certain number of blocks at once (similar to ARMORED [11]) to avoid re-computations for every block.

6. CONCLUSION

We presented the design, implementation and evaluation of RamCrypt, a Linux kernel patch that transparently encrypts the address spaces of user mode process under Linux. RamCrypt effectively protects against memory disclosure attacks that give an attacker access to the physical memory and a variety of physical attacks on RAM such as cold boot and DMA attacks. RamCrypt can be easily enabled on a per-process basis without the need for binary rewriting or recompilation, and it is fully compatible to existing systems. If enabled for a single-threaded process with a sliding window size of four, which is a reasonable choice regarding the tradeoff between security and performance, it slows processes of the sysbench benchmarking suite down by 25%.

To the best of our knowledge, RamCrypt is the first solution that transparently encrypts the address space of user processes within the Linux kernel while being fully compatible to existing applications and storing the key outside RAM.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) and by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA).

7. REFERENCES

[1] Root exploit on Exynos. http://forum.xda-developers.com/showthread.php?t=2048511.

[2] Bastian Reitemeier. Palinopsia: Reconstruction of FrameBuffers from VRAM. https://hsmr.cc/palinopsia/, Mar. 2015.

[3] Becher, M., Dornseif, M., and Klein, C. N. FireWire - All Your Memory Are Belong To Us. In *Proceedings of the Annual CanSecWest Applied Security Conference* (2005).

[4] Blass, E., and Robertson, W. TRESOR-HUNT: attacking cpu-bound encryption. In *28th Annual Computer Security Applications Conference, ACSAC, Orlando, FL, USA* (2012), pp. 71–78.

[5] Boileau, A. Hit by a Bus: Physical Access Attacks with Firewire. In *Proceedings of Ruxcon '06* (Sydney, Australia, Sept. 2006).

[6] Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 321–336.

[7] Duc, G., and Keryell, R. Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In *22nd Annual Computer Security Applications Conference ACSAC 2006), Miami, Florida, USA* (2006), pp. 483–492.

[8] Dunn, A. M., Lee, M. Z., Jana, S., Kim, S., Silberstein, M., Xu, Y., Shmatikov, V., and Witchel, E. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI* (2012), pp. 61–75.

[9] Garmany, B., and Müller, T. PRIME: private RSA infrastructure for memory-less encryption. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA* (2013), pp. 149–158.

[10] Götzfried, J., and Müller, T. Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption. In *Transactions on Information and System Security (TISSEC)*, vol. 17.

[11] Götzfried, J., and Müller, T. ARMORED: cpu-bound encryption for android-driven ARM devices. In *International Conference on Availability, Reliability and Security ARES* (2013), pp. 161–168.

[12] Guan, L., Lin, J., Luo, B., Jing, J., and Wang, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *36th IEEE Symposium on Security and Privacy* (2015).

[13] Gutmann, P. Data remanence in semiconductor devices. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA* (2001).

[14] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest We Remember: Cold Boot Attacks on Encryptions Keys. In *17th USENIX Security Symposium* (2008).

[15] Harrison, K., and Xu, S. Protecting cryptographic keys from memory disclosure attacks. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007* (2007).

[16] Henson, M., and Taylor, S. Beyond full disk encryption: Protection on security-enhanced commodity processors. In *Applied Cryptography and Network Security ACNS* (2013), pp. 307–321.

[17] Jürgen Pabel. Frozen Cache. http://frozenchache.blogspot.com/, Jan. 2009.

[18] Kannan, J., and Chun, B. Making programs forget: Enforcing lifetime for sensitive data. In *13th Workshop on Hot Topics in Operating Systems, HotOS XIII, Napa, California, USA, May 9-11, 2011* (2011).

[19] Latham, D. C. *Department of Defense trusted computer system evaluation criteria.* 1985.

[20] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy HASP* (2013).

[21] Müller, T., Freiling, F., and Dewald, A. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security Symposium* (Aug. 2011).

[22] Müller, T., Taubmann, B., and Freiling, F. C. Trevisor - os-independent software-based full disk encryption secure against main memory attacks. In *Applied Cryptography and Network Security ACNS, Singapore* (2012), pp. 66–83.

[23] Patrick Simmons. Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption. *CoRR abs/1104.4843* (2011).

[24] Peterson, P. Cryptkeeper: Improving security with encrypted RAM. In *Technologies for Homeland Security (HST)* (Nov 2010), pp. 120–126.

[25] Provos, N. Encrypting virtual memory. In *9th USENIX Security Symposium* (2000).

[26] Reardon, J., Basin, D. A., and Capkun, S. On secure data deletion. *IEEE Security & Privacy (Oakland) 12*, 3 (2014), 37–44.

[27] Skorobogatov, S. P. Data remanence in flash memory devices. In *Cryptographic Hardware and Embedded Systems CHES* (2005), pp. 339–353.

[28] Stewin, P., and Bystrov, I. Understanding DMA malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment DIMVA* (2012), pp. 21–41.