Isolating Operating System Components with Intel SGX

Lars Richter, Johannes Götzfried, and Tilo Müller
Department of Computer Science
FAU Erlangen-Nuremberg
{lars.richter,johannes.goetzfried,tilo.mueller}@fau.de

ABSTRACT

In this paper, we present a novel approach on isolating operating system components with Intel SGX. Although SGX has not been designed to work in kernel mode, we found a way of wrapping Linux kernel functionality within SGX enclaves by moving parts of it to user space. Kernel components are strictly isolated from each other such that a vulnerability in one kernel module cannot escalate into compromising the entire kernel. We provide a proof-of-concept implementation which protects an exemplary kernel function, namely full disk encryption, using an Intel SGX enclave. Besides integrity of the disk encryption, our implementation ensures that the confidentiality of the disk encryption key is protected against all software level attacks as well as physical attacks. In addition to the user password, we use a second authentication factor for deriving the encryption key which is stored sealed and bound to the platform. Thus, stealing the hard drive and sniffing the user password is insufficient for an attacker to break disk encryption. Instead, the two factor authentication scheme requires an attacker to additionally obtain the actual machine to be able to break encryption.

CCS Concepts

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

Keywords

Intel SGX, Linux Kernel, Isolation

1. INTRODUCTION

The protection of computer systems against malicious applications and the isolation of software components is still a difficult task for software developers, software architects and researchers. Many attempts have been made to secure modern systems but the rising complexity and dependencies of big applications allow highly sophisticated attacks to succeed. If one software component within a given system is compromised, it is often easy to compromise the other components with the help of privilege escalation. This is especially true for compromised software components *within* the operating system as one compromised operating system component can easily compromise other components of the kernel without the need for further vulnerabilities. To enforce isolation of operating system components even in the event of a partially compromised kernel, trust needs to be rooted, for example, in hardware.

In 2013, Intel published the Intel *Software Guard Extensions* (SGX) in a series of papers [17, 1]. With SGX, software can be executed within containers, so called *enclaves*, which are shielded from all other software running on the system including privileged software such as the operating system or the hypervisor. The Intel Skylake processor, released in October 2015, allows the creation and execution of SGX enclaves via its extended instruction set. Various researchers already published their ideas and proof of concepts of how applications can benefit from SGX [2, 20].

For most proposals it is assumed that the system is initially uncompromised and that an attacker enters the system from the outside, for example, from the network interface or other peripheral devices of the computer. Furthermore, even if one malicious application is executed on the system, it is assumed that at least the operating system can be trusted. Since operating systems have a huge code base these days and vulnerabilities in operating systems are quite common, this assumption does not necessarily hold true.

The security model of Intel SGX states that only the CPU needs to be trusted and all software including applications and the operating system is considered untrusted. Secured containers (trusted enclaves) are protected by hardware mechanisms and no malicious application, operating system or virtual machine monitor can access the information in the secured container. Furthermore, the memory belonging to these containers is transparently encrypted by the CPU to defend against physical hardware attacks and enclaves can attest their current state to a remote party to prove that they have not been modified during loading.

Given the security model of SGX with its strong security guarantees and protection mechanisms against potentially malicious high-privileged system software, SGX looks like a promising hardware extension to shield kernel components against each other and enforce isolation of specific parts within an operating system. However, SGX enclaves can only be entered from ring three (user mode) and not ring zero (kernel mode). Consequently, it is not directly possible to put kernel functionality within SGX enclaves and to use this functionality from non-protected parts of the kernel.

SysTEX '16, December 12-16 2016, Trento, Italy

ISBN 978-1-4503-4670-2/16/12…$15.00

DOI: http://dx.doi.org/10.1145/3007788.3007796

To still be able to provide isolation for operating system components, we developed a solution for moving parts of an operating system kernel to user space and protecting those parts with the security features provided by Intel's SGX.

1.1 Our Contribution

In this paper, we present a generic concept to provide SGX security guarantees for operating system components. Our concept provides tamper resistance for kernel functionality and is able to strictly isolate operating system components from each other. To the best of our knowledge, we are the first showing that Intel SGX can be used to secure kernel functionality. In detail our contributions are:

- Instead of shielding user mode applications from other user mode applications, we leverage SGX to secure Linux kernel modules. Similar to a microkernel, kernel components are isolated from each other such that a vulnerability in one kernel module cannot escalate into compromising the kernel. Furthermore, our solution prevents privileged malicious user space applications from attacking secured kernel modules.
- Because SGX has not been designed to work in kernel mode, we present a generic concept to move parts of the kernel to the user space and then protect the given functionality by wrapping it in SGX enclaves.
- We provide a proof-of-concept implementation of our concept which protects one kernel function, namely full disk encryption, using an Intel SGX enclave in user space. Our implementation is seamlessly integrated by using the Linux Crypto API; the integrity of the disk encryption as well as the confidentiality of the encryption key is protected against all software level attackers. In addition, the disk encryption key is also protected against physical attacks.
- Compared to usual disk encryption solutions, our prototype derives the encryption key not only from a user password but also from a second factor bound to the machine. This factor is stored sealed by the platform such that it can only be used indirectly by a software attacker on that same platform. Retrieving the user password and stealing the hard drive is insufficient for breaking disk encryption.
- We evaluated our prototype regarding correctness, security, and performance. Although our solution is a step forward in terms of security, it imposes a notable performance overhead up to factor 100.

Our implementation is free software published under the GPL v2. It is available at `https://www1.cs.fau.de/sgx-kernel`.

1.2 Related Work

Basic isolation concepts, like horizontal isolation of the system layer against applications, and vertical isolation of applications against each other, are available in modern operating systems since decades [23]. These basic isolation concepts are supported by hardware extensions like an MMU/MPU and CPU protection rings. To guarantee tamper resistance of software components also in the presence of attackers with system level privileges, however, stronger degrees of isolation are required. These degrees of isolation can only be provided by new hardware extensions that have an immutable trust anchor, such as the *Trusted Platform Module* (TPM) [24] which is deployed, for example, by Flicker [16].

Today the TPM is widely deployed as dedicated chip, as it does not rely on CPU modifications. The security guarantees are weaker compared to SGX because not only the CPU packages but also the TPM chip and buses are considered trusted. As the *Trusted Computing Base* (TCB) consists of the whole software stack including the operating system kernel, using the TPM for protecting only parts of an application or the operating system is impracticable. Hence, the TPM today is almost only used at boot time, for example, by Microsoft BitLocker.

To overcome the restriction of all software having to be part of the TCB, Intel introduced the *Trusted Execution Technology* (TXT), which also uses the TPM, but additionally allows to dynamically establish a new root of trust for software running in a virtualized environment besides the software stack. TXT ensures that the virtualized software has exclusive control over the device by suspending all other software, i.e., the OS and all running applications. However, suspending all other software of a device for the TXT software to run, negatively impacts performance and might even lead to losing interrupts depending on its size. Furthermore, TXT does not provide RAM encryption and is thus prone to physical attacks on main memory. Fides [22] and TrustVisor [15] are academic prototype implementations which build on the functionality of Intel TXT.

Besides the commercial trusted computing solutions SGX, TPM, and TXT, there are numerous academic proposals for small embedded devices [19, 10, 14] as well as for general purpose computers [5, 7]. So far, Intel SGX has been used in an untrusted cloud context [2, 20]. To the best of our knowledge, however, there is currently no solution which leverages SGX to isolation for operating system components.

2. BACKGROUND: INTEL SGX

Just recently, Intel provided an open source version of the Intel SGX SDK for Linux which we used for our proof-of-concept implementation. For a detailed description of SGX itself, please refer to the official Intel SGX Instruction Set Reference manual [12]. Furthermore, there is an exhaustive summary paper by Costan and Devadas [6] which we highly recommend for a deep understanding of SGX internals.

The SGX software and hardware stack consists of multiple parts as shown in Figure 1. The basis builds an SGX-enabled CPU with an extended instruction set and memory access mechanisms. These instructions are used to create, launch, enter and exit an enclave. Memory for an enclave is allocated within the *Enclave Page Cache* (EPC) which is part of the *Processor Reserved Memory* (PRM) and protected with the help of a *Memory Encryption Engine* (MEE).

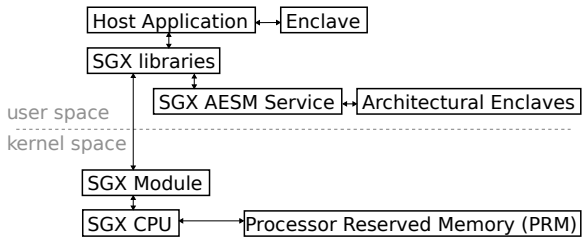

Figure 1: High-level overview of the SGX hardware and software stack.

The host application is an untrusted component. The untrusted host application can call trusted functions inside the enclave. Neither the input to the enclave, nor the output of the enclave can be fully trusted because a malicious operating system could modify inputs and outputs of the enclave. Thus, the enclave author has to be aware of this fact and, for example, take care of properly sanitizing enclave input data.
To initiate the enclave, a launch token is needed which can be retrieved with the help of Intel's *Launch Enclave* (LE). For convenience, the access to the Launch Enclave and other architectural enclaves such as the *Quoting Enclave* (QE) and the *Provisioning Enclave* (PE) is provided by the Intel AESM service in user space. SGX libraries provide necessary functionality to communicate with the AESM service.
As stated previously, enclaves can only be entered in user space. However, creating and initiating an enclave is only possible from kernel space. Therefore, a privileged SGX kernel module must be installed on any system providing SGX for user mode applications. The SGX kernel module takes care of managing the EPC and dispatching specific SGX instructions needed for enclave initialization.

3. DESIGN AND IMPLEMENTATION

In this section, we show how isolation between Linux kernel components can be enforced with the help of SGX.

3.1 Architecture

As described in Section 1, it is not possible to enter an enclave from kernel space. An enclave's code has always to be executed in ring three with a reduced set of allowed instructions and a limited amount of available memory. Furthermore, it is not possible to initiate an enclave on its own but instead the Intel Launch enclave must be used to generate the appropriate launch token.
To overcome these major limitations of SGX, we decided to build an architecture which moves part of the kernel functionality to user space such that the core functionality can then be wrapped by an enclave. This enclave is implemented by a user space service or daemon which calls the Intel Launch enclave for initialisation. Once the enclave is running, functionality within the enclave can be used by the daemon. Consequently, the kernel first has to communicate with the daemon which then passes the request to the enclave.

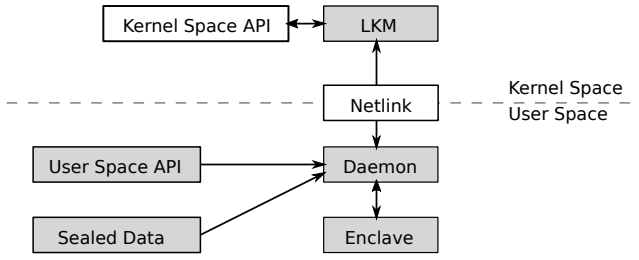

Figure 2: An LKM using functionality wrapped by an enclave. The LKM communicates with a user mode daemon which forwards the requests to the enclave residing in user space.

Figure 2 shows the architecture which enables Linux kernel modules to use functionality wrapped by SGX enclaves. On the one hand, the LKM uses the API of the Linux kernel to provide a certain functionality within the kernel. On the other hand, the LKM communicates with a user space daemon via a Netlink interface to send certain requests or receive responses. The daemon just forwards the requests and responses to and from the enclave and thus, the enclave is able to provide functionality to the LKM which the LKM itself provides to the kernel. In addition, the daemon is allowed to interact with other user space applications if necessary and stores sealed data of the enclave. Consequently, the enclave and thus the LKM is able to securely store persistent data across reboots.
To start and initialize the daemon, the LKM uses the user mode helper API of the Linux kernel. After the daemon has been started, it informs the kernel about its state and then data can be transferred.

3.2 Communication

During our work, we tested different communication protocols for exchanging data between kernel and user space. Netlink interfaces have the advantage that they can be easily used by kernel modules and no patching of the kernel itself is required. Furthermore, it is possible to implement callback functions on incoming Netlink messages in kernel and user space, thus avoiding polling. A drawback of the Netlink communication is reduced throughput. Nevertheless, Netlink interfaces are used in this work because a reliably bidirectional connection is more important than maximal performance.

3.3 Prototype Implementation

We provide a prototype implementation of our architecture and use it in the scope of full disk encryption. Figure 3 shows a specialized version of Figure 2. Our LKM registers a new cipher within the crypto API of the Linux kernel which can then be used by dm-crypt. The encryption algorithm used for full disk encryption is implemented within an enclave, and thus it is guaranteed that the implementation cannot be tampered with. The key used for disk encryption is securely derived within the enclave from a password chosen by the user and a device specific salt. The user password can be entered with the help of a tool which communicates with the daemon directly in user mode and the salt is stored sealed to the enclave identity. Consequently, it cannot be unsealed on a different device.

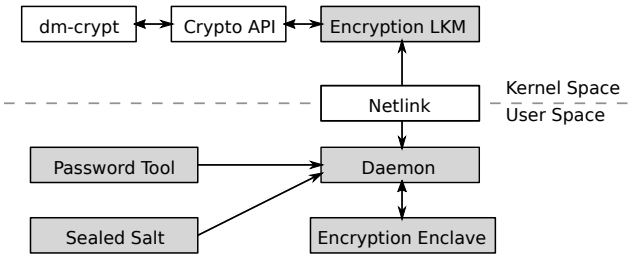

Figure 3: Our generic architecture used in the scope of full disk encryption. The encryption algorithm is securely executed within an enclave.

Full disk encryption is a well-known procedure to guarantee the confidentiality of sensitive data even in the event of attackers having physical access to a computer system. When the device is turned off, the data is at rest and no cryptographic keys are stored on the device. When it is turned

on, however, the user must insert a passphrase to initialize the cryptographic cipher. For almost all full disk encryption solutions, the password and the resulting key are stored in clear within main memory. Our implementation, however, stores the key securely within an enclave and protects it by logical and physical means provided by SGX.

3.4 Protection against Physical Attacks

Besides isolation, we also provide protection against physical attacks which is especially useful in the context of full disk encryption. Currently, there are only very few disk encryption solutions which protect the disk encryption key from physical attacks on main memory. TRESOR [18, 9] and ARMORED [8] use the debug registers of Intel and ARM processors instead of RAM to store the encryption key and all intermediate states of the encryption algorithm. Consequently, sensitive key-related data is not stored in RAM and cannot be retrieved with physical hardware attacks.
Multiple attacks on sensitive data in main memory exist and some can be used against TRESOR and ARMORED. Cold boot attacks on RAM allow the retrieval of encryption keys after the system is turned off [11]. Sensitive data from main memory can also be obtained using DMA attacks, for example, via Firewire [4] or USB OTG [13]. TRESOR and ARMORED generally prevent against cold boot attacks, but not DMA attacks, because the control flow of the kernel can be changed to extract keys [3].
In contrast to TRESOR and ARMORED, our solution not only protects against cold-boot attacks, but also against DMA attacks, because SGX ensures that no peripheral device is able to read enclave memory. Furthermore, the use of a sealed salt adds another factor to the encryption key generation. Even if the user password could be retrieved from main memory, it is still not sufficient to derive the actual encryption key because the salt as well as the derived key is never stored outside the enclave in clear.

3.5 Detailed Workflow

The overall functionality of our implementation is spread between the LKM and the user space daemon. In the following, the initialisation sequence, the key setting process, and the encryption and decryption functionality will be described.

Initialising LKM and daemon.
When the kernel module is initialized, it registers a Netlink family to communicate with the daemon. Once the Netlink socket is created, it starts the daemon via the user mode helper API. The daemon then creates and starts the enclave.

Setting password using the daemon.
Because using the key setting functionality of the crypto API would leak the key or password to main memory, we provide a possibility to directly set the password using only the daemon. To this end, the daemon opens a named pipe and waits until the password is written to that pipe.

Deriving disk encryption key.
After the password has been read from the user, the daemon loads a predefined file from disk which contains the sealed salt. The enclave checks if the sealed salt is valid and unseals it. If the sealed data is not valid, it will generate a new salt and seal it. PBKDF2 is used to finally derive the disk encryption key from the user password and the salt.

Establish Netlink communication.
The daemon creates the same Netlink interface as the kernel module and sends an *initialization succeeded* message to the kernel. The kernel receives the message and registers the new cipher at the crypto API.

Data encryption and decryption.
After initialization, the encryption and decryption process are straight forward. The encrypt and decrypt callback functions of our LKM are called by the user of the crypto API. The LKM then sends a Netlink message to the daemon, which calls the encrypt and decrypt functions of the enclave. The enclave performs the requested cryptographic operation and returns the encrypted or decrypted block which is passed back to the kernel via Netlink. Finally, the kernel module copies the block to the destination given by the caller of the crypto API and returns.

3.6 Enclave Cryptography

To register a new cipher within the crypto API, at least routines for encryption and decryption need to be provided. Usually also a key setting routine is required, but because we set the key using our daemon directly, we do not need to implement this routine for the crypto API.
The easiest way to provide a new cipher within the crypto API is to register a bare block cipher such as AES, because it allows the crypto API to combine the bare block cipher with all modes of operation that are already implemented in the kernel. Intel provides a cryptography library together with th SGX SDK, but it is only possible to use AES together with preselected modes of operation. We therefore decided to use a small Intel AESNI library which provides a standalone bare block cipher implementation of AES with multiple key sizes and uses AESNI, Intel's hardware AES instructions.

4. EVALUATION

In this section, our prototype implementation is evaluated regarding correctness (Section 4.1), performance (Section 4.2), and security (Section 4.3).

4.1 Correctness

The correctness of our implementation regarding the encryption and decryption functionality has been tested in different ways. The crypto API of the kernel provides a test manager which initializes a given cipher, sets a predefined key, and then checks encrypted and decrypted blocks with different sizes against predefined test vectors. Only if the resulting blocks are equal to the ones stored within the test vectors, the implementation is correct.
When using PBKDF2 with the user password and the salt as input, the data will be encrypted with a different key compared to the user password being directly passed to the crypto API. To still be able to test the functionality of our implementation, the enclave was temporarily modified to print the PBKDF2 derived key to the host application. Once this key was fed into the crypto API directly, we could verify that encrypted and decrypted blocks are equal and thus our implementation is correct.
Furthermore, we were able to encrypt and decrypt partitions over multiple reboots and thus verified that the key can always be derived on the computer using the same user password and sealed salt.

4.2 Performance

We compared our implementation to the standard AES implementation of the Linux kernel in a full disk encryption scenario. In addition we compared both variants to a non-encrypted disk access. All tests have been performed on a Dell Inspiron 7559 notebook with an Intel i7-6700HQ CPU, sixteen gigabytes of main memory, and a Seagate ST1000LM024 hard drive. We used a vanilla version of the Ubuntu 15.10 distribution running Linux kernel version 4.4.7. During the design, it became clear that the multi-layered approach will result in many context switches which decrease performance. Our performance tests show that this assumption is true. In Table 1, three different performance tests with our implementation, the default Linux AES implementation and unencrypted disk access is shown. Three different partitions were mounted on the same hard disk for this evaluation and 24 tests have been executed before calculating the median of the results. The Linux command line tool `dd` was used to analyse the write speeds and `hdparm` for measuring the read speeds.

Test	Plain	AES	SGX
dd 100mb block write	107.0	104.5	1.1
hdparm uncached read	110.1	113.7	1.1
hdparm cached read	13,289.5	12,004.3	1,576.7

Table 1: Reading and writing speeds in MB/s of different operations with plain disk access, the default AES implementation, and our implementation.

In detail, `dd` was executed with the parameters `fdatasync` and `notrunc` using a block size of 100 megabytes. This results in a physical non-truncated write of one file with the size of exactly 100 megabytes. After `dd` has completed its operation, the average write speed for that operation is printed.
The tool `hdparm` was executed with both, the `-t` and `-T` option. The first option performs uncached disk reads and the buffer cache is cleared before performing the read. The second option performs cached reads and displays the reading speed from the Linux buffer cache without forced disk access. Our implementation reaches about 1% of the read and write performance with forced disk access and about 10% of the read performance using the buffer cache. To write one encrypted block to disk, the kernel must send the data block over the Netlink bus to the daemon. The daemon must enter the enclave, which is another context switch, and then the enclave is able to encrypt the block using AESNI instructions. In return, the enclave sends the encrypted block back over the Netlink bus via the daemon to the LKM. Our implementation is meant to be a proof-of-concept for demonstrating that isolating kernel components using SGX is possible and improving the performance is part of future work.

4.3 Security

In this section, we first show why our design prevents wrapped kernel components from being tampered with. Second, we evaluate the security of our prototype implementation regarding physical hardware attacks.
Functionality moved to user space and wrapped by an enclave as shown in Section 3 is protected against all software attackers due to the strong security guarantees of Intel SGX. Even if the kernel is compromised, the kernel component within the enclave cannot be tampered with because accesses are prevented by protection mechanisms in hardware. Even though SGX does not support entering enclaves in ring zero, we found a way to protect ring zero functionality with SGX. Our prototype implementation derives the disk encryption key within an enclave. At no point during the whole enclave lifecycle, the key is passed to the outside world. Furthermore, the salt is generated randomly and unknown outside the enclave as well. Consequently, any attempt to retrieve the key or the salt by physically acquiring main memory will fail. Of course, it is still possible to physically attack a system running our prototype and, to use a copy of it as a black box to encrypt and decrypt data. For the attack to succeed, however, an attacker must obtain all of the following components of the cryptographic system:

- user password
- sealed salt
- unmodified enclave which sealed the salt
- CPU on which the salt was sealed

The user password and the salt are needed as input for PBKDF2 to derive the encryption key. The salt can only be unsealed with the very same enclave on the same CPU, because it is bound to the enclave and the CPU. Consequently, it is not sufficient to steal the user password which makes our solution practically stronger than TRESOR and ARMORED, where it is sufficient to steal the password. We also verified that the user password which at some point has to be entered by the user and sent to daemon does not reside in RAM by using the *Linux Memory Extractor* (LiME)[1].
Contrary to the user password, the *sealed* salt can be retrieved through a physical hardware attack at any time. Even though, the sealed salt cannot be unsealed without possessing the actual device, it is recommended storing the sealed salt on a removable storage device such as a thumb drive and only connect it to the computer if the encrypted partition needs to be mounted. This way, the computer needs to be stolen, the password needs to be retrieved, and also the removable storage device needs to be acquired to finally be able to decrypt sensitive data.

5. CONCLUSION AND FUTURE WORK

In this section, we give an outlook over future research directions and conclude our work.

5.1 Future Work

Although our solution works quite well as a prototype, performance must be improved in future versions of the implementation. There is a lot of research comparing the throughput of the Linux Netlink interface to other communication channels like, for example, *SYS V Message Queues* and *SYS V Shared Memory* [21]. It has been shown that compared to the SYS V Message Queues, the Netlink user-to-kernel communication is around 30%, the kernel-to-user communication around 55%, and the startup time around 66% slower in terms of processor cycles. SYS V shared memory has a higher throughput, but needs longer startup times than message queues.
Single block cryptography leads to small chunks of data being sent over the Netlink interface and increasing the blocksize would result in a better throughput of the system. This, however, requires implementing the modes of operation within the enclave or at least the daemon.

[1] https://github.com/504ensicsLabs/LiME

With the current implementation it is not possible to encrypt all partitions including the root partition of a system, because the daemon must be executed in user space and loaded from some partition first. Future implementations can address this problem by providing a solution where the daemon is loaded from a different location like an initialization ramdisk.

5.2 Conclusion

In this paper, we showed a concept to enforce isolation for kernel components by moving parts of their functionality to the user mode and wrapping it within Intel SGX enclaves. To this end, we require a daemon running in user mode communicating with an LKM in kernel mode. In addition to the LKM interacting with the kernel API, the daemon is also allowed to directly provide functionality to user space. As a proof-of-concept implementation, we wrapped disk encryption functionality within an enclave and provide the resulting cipher via the crypto API of the Linux kernel. The prototype not only isolates the disk encryption implementation from other parts of the kernel, but also the disk encryption itself is hardened by a second factor in form of the machine which has to be present for deriving the same disk encryption key. During runtime, the disk encryption key as well as the salt is never released to unprotected main memory, thus protected against physical hardware attacks.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

6. REFERENCES

[1] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), p. 10.

[2] Baumann, A., Peinado, M., and Hunt, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014).

[3] Blass, E.-O., and Robertson, W. Tresor-hunt: attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM, pp. 71–78.

[4] Böck, B., and Austria, S. B. Firewire-based physical security attacks on windows 7, efs and bitlocker. *Secure Business Austria Research Lab* (2009).

[5] Champagne, D., and Lee, R. B. Scalable Architectural Support for Trusted Software. In *Proceedings of the 16th International Conference on High-Performance Computer Architecture* (2010).

[6] Costan, V., and Devadas, S. Intel sgx explained. Tech. rep., Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org.

[7] Costan, V., Lebedev, I., and Devadas, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. *IACR Cryptology ePrint Archive*, 564 (2015).

[8] Götzfried, J., and Müller, T. ARMORED: cpu-bound encryption for android-driven ARM devices. In *2013 International Conference on Availability, Reliability and Security (ARES'13)*, pp. 161–168.

[9] Götzfried, J., and Müller, T. *Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption*, 2 ed., vol. 17. New York, USA, 2014.

[10] Götzfried, J., Müller, T., de Clercq, R., Maene, P., Freiling, F., and Verbauwhede, I. Soteria: Offline Software Protection Within Low-cost Embedded Devices. In *Proceedings of the 31st Annual Conference on Computer Security Applications* (2015).

[11] 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 encryption keys. *Communications of the ACM 52*, 5 (2009), 91–98.

[12] Intel. *Intel Software Guard Extensions Programming Reference*, 329298-002us ed., October 2015.

[13] Jodeit, M., and Johns, M. Usb device drivers: A stepping stone into your kernel. In *European Conference on Computer Network Defense (EC2ND)* (2010), IEEE, pp. 46–52.

[14] Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the Ninth European Conference on Computer Systems* (2014).

[15] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 31st Symposium on Security and Privacy* (2010).

[16] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An Execution Infrastructure for Tcb Minimization. In *3rd European Conference on Computer Systems* (2008).

[17] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[18] Müller, T., Freiling, F. C., and Dewald, A. Tresor runs encryption securely outside ram. In *USENIX Security Symposium* (2011).

[19] Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *22nd USENIX Conference on Security* (2013).

[20] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015).

[21] Slominski, R. *Fast User/Kernel Data Transfer*. PhD thesis, College of William & Mary, 2007.

[22] Strackx, R., and Piessens, F. Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware. In *Proceedings of the 19th Conference on Computer and Communications Security* (2012).

[23] Tanenbaum, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, NJ, USA, 2008.

[24] Trusted Computing Group. *TPM Main: Part 1 Design Principles*, Version 1.2, Revision 116 ed., 2011.