Securing Runtime Memory via MMU Manipulation

Marinos Tsantekidis
Institute of Computer Science - FORTH
Heraklion, Greece
E-mail: tsantekid@ics.forth.gr

Vassilis Prevelakis
AEGIS IT RESEARCH GmbH
Braunschweig, Germany
E-mail: vp2020@aegisresearch.eu

***Abstract*—It is often useful for a code component (e.g., a library) to be able to maintain information that is hidden from the rest of the program (e.g., private keys used for signing, or usage counters used for behavioral monitoring of the program). In this paper, we present an extension to a previously developed mechanism for controlling access to libraries, in order to implement a scheme that allows each library to have its own private storage space. When running code outside the address space of a given library, the pages containing the private memory of that library are not mapped into the program's address space, hence are not accessible to the rest of the program. Finally, we present an API that allows library developers to utilize private storage.**

***Keywords*—Secure; Run-time; Memory; MMU.**

I. Introduction

The advancement of technology is everlasting and non-stop, which leads to modern software systems becoming more and more complex. This results in new challenges and vulnerabilities being discovered every day and users increasingly requiring security considerations and provisions for their applications. At the same time, there is a parallel and oftentimes one-step-ahead increase in attackers' capabilities and effectiveness, especially if there is profit involved in their illicit activities. However, complete security of a program is unfeasible. Conceding that vulnerable code will be included in production software systems, there is a need to either detect these vulnerabilities so that they may be fixed before an adversary can exploit them in a zero-day attack or determine if such a vulnerability is actively being exploited. Our compromise is that by monitoring the behavior of a program we can distinguish such situations, determine whether their cause is security-related and, if so, take appropriate corrective actions. We implement such actions at an abstract level, between the Operating System (OS) and a running application. Our approach is to break up a running application into its main components (essentially the main program and the libraries it uses) by leveraging the Memory Management Unit (MMU) of the Linux kernel and examine the interactions between the individual components. We use two techniques for our analysis, based our previous work [1]–[3], which enables us to intercept all library calls from both the user as well as the kernel side, analyze them and take some form of action (reporting, argument checking, policy enforcement, etc.) before allowing them to continue.

Looking at the subject of software run-time behavior monitoring, analysis and modification from another point of view, we propose to implement the notion of a Trusted Execution Environment (TEE) at the memory space of a user application. A TEE [4] is a secure, integrity-protected processing environment, consisting of memory and storage capabilities [5]. It establishes an isolated execution environment that runs parallel to a standard OS and it protects sensitive code and data from privileged attacks without compromising the native OS. It prevents unauthorized access or modification of executing code and data while they are in use, so that the applications running the code can have high levels of trust in the TEE, because they can ignore threats from the rest of the system. Hardware vendors (e.g., Intel) have already implemented the concept of TEE into their products (e.g., SGX technology). Virtual TEEs (e.g., Open-TEE [6]) allow developers to create trusted applications using the GlobalPlatform TEE specification [7].

In this paper, we present our idea to include the concept of a TEE to our previous work [2] [3], where we program the MMU in such a way so as to map protected private pages into the address space of a running program, that are accessible only by specific functions inside the external libraries that said program uses. Upon interception of a library call, our system - after redirecting the call through the *gate* (already mapped, specially crafted library) - determines if the call can access the information stored securely in the newly-mapped private memory. In this way, we protect sensitive data inside a secure enclosure and we minimize what can access them, as we limit their exposure to only a specific set of legitimate functions found in the *gate* library, imposing serious limitations on what actions can be performed on the protected data, by what part of the program and at which point in execution time.

The remainder of this paper is organized in the following manner: In Section II, we present some important work that has been carried out over the years with respect to defenses against code injection/reuse attacks C[IR]As, as well TEEs - the two aspects of our approach. In Section III, we detail the design of our mechanism. In Section IV, we describe the implementation specifics. In Section V, we evaluate our approach both in terms of performance and memory coverage. We also list two real-life scenarios where our mechanism can be used. In Section VI, we conclude our work.

II. Background and related work

Behavior control techniques have been the subject of research against code reuse attacks [8]–[13] for many years.

The DisARM defense technique [14] protects against both code-injection and code-reuse based buffer overflow attacks

 ISBN: 978-1-61208-919-5

by breaking the ability of attackers to manipulate the return address of a function. DisARM uses a fine-grained analysis of the binary to find all critical interactions that manipulate the hardware PC and verifies any change to the PC before the change is applied. For each such critical instruction, a verification block is inserted immediately before the instruction in order to evaluate whether the target address is valid with respect to the current instruction the program is executing.

Kanuparthi et al. [15] propose a hardware-based dynamic integrity checking approach. It permits the instructions to commit before the integrity check is complete, and allows them to make changes to the register file, but not the data cache. The changes made by the instructions are held in the store buffer or in a shadow register file until the check is complete. Then, the values are accordingly written to the L1 data cache or the original register file. The system is rolled back to a known state, if the checker deems the instructions as modified.

In [16], Graziano et al. discuss a new class of Direct Kernel Object Manipulation (DKOM) attacks that they call Evolutionary DKOM (E-DKOM). The goal of this attack is to alter the way some data structures "evolve" over time. It targets the evolution of a data structure in memory, with the goal of tampering with a particular property of the operating system. On the attack side, they are able to temporarily block any process or kernel thread, without leaving any trace that could be identified by existing DKOM detection and protection systems. On the defense side, they present the design and implementation of a hypervisor-based detector that can verify the fairness of the OS scheduler. Their implementation shows that it needs to be customized on a case-by-case basis and that evolutionary attacks are very hard to deal with, requiring more research to mitigate this threat.

Kayaalp et al. [17] examine a signature-based detection of code reuse attacks (CRAs), where the attack is detected by observing the behavior of programs and detecting the gadget execution patterns. They demonstrate a new attack that renders previously proposed signature-based approaches ineffective by introducing delay gadgets, in order to obfuscate the execution patterns of the attack without performing any useful computation. They develop a complete working JOP attack that incorporates delay gadgets. Then, they propose and develop the Signature-based CRA Protection (SCRAP) hardware-based architecture for detecting such stealth JOP attacks. SCRAP recognizes the formal grammar that expresses the attack signatures or the patterns of executed instructions that are indicative of a JOP attack, which are significantly different from those of the regular programs as they execute frequent indirect *jump* (or *call*) instructions to jump from gadget to gadget.

Additionally, with regards to TEEs, Intel's Software Guard Extensions (SGX) [18] is a hardware feature that helps encrypt a portion of memory. This portion - *enclave* - is used by the OS/applications to define private regions of code and data that cannot be accessed by any (potentially running at a higher privilege level) process outside the enclave, thus preserving the confidentiality and integrity of sensitive code and data. However, several attacks have been developed that break the security of SGX. In [19], Schwarz et al. were able to extract a full RSA private key by performing a cache side-channel attack on a co-located SGX enclave. Later on, countermeasures were released against this attack [20] [21]. More recently, the Spectre attack [22] was adapted to target SGX enclaves [23]. Similarly, the Foreshadow attack exploits speculative execution (e.g., Spectre) in order to read the contents of SGX-protected memory [24]. Additionally, it has been proven that a ROP attack can be constructed and launched all from within an enclave [25] [26]. However, a defense against this attack vector was later presented in [27].

III. Design

The goal of our proposed approach is two-fold. On one hand, since it is based on our previous approach [2] [3], it thwarts control-flow hijacking attacks by segregating a process's executable areas which correspond to its external libraries or the main executable. It, then, imposes strict control over any attempt to invoke such an area, by redirecting all calls through a *gate* library - mapped by a custom Linux kernel, one for each area - where we can implement several checks before allowing a call to move forward (Figure 1).

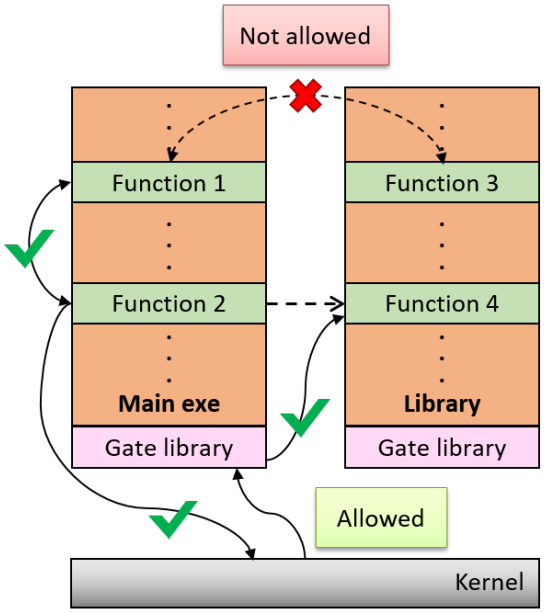


Figure 1. Memory segmentation and access control

On the other hand, it protects sensitive information of an application (e.g., a private signing key) by mapping private secure memory pages for each area at run-time and making them accessible only to specific functions inside the *gate* library and only at specific intervals during execution (Figure 2).

Separation of data used by the libraries from data used by the running application is a significant step of our approach. Originally, the application and library code share their stack and heap spaces, which provides a breeding ground

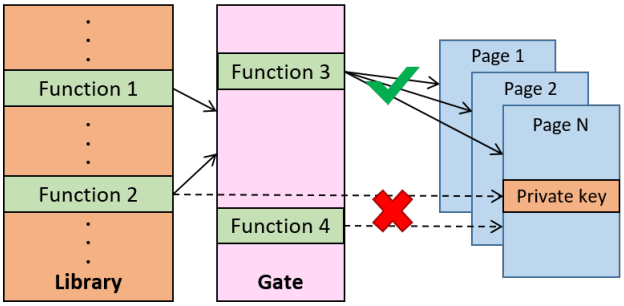


Figure 2. Secure memory mapping

for interfering with the execution of library code. Since we already have a mechanism that allows us to rewrite the page table whenever a library boundary is crossed, we can now extend it by adding private memory for every library. This is memory that is accessible only when running code of a specific library; code outside this library will find the pages associated with the private memory inaccessible. In this way, our *gates* can maintain state (e.g., which library tried to access the gate indicating a possible breach attempt if different from the associated one, how many times a given routine has been called, or the sequence of calls to various library functions). Library code can, thus, take advantage of private memory to protect its own data structures e.g., making them completely inaccessible (no read/write/execute rights) to the rest of the program.

Transparency is, also, of paramount importance. Applications continue to work as originally intended by the developer, but the access control mechanism underneath delivers secure execution of the program. When a call to a separated library is intercepted, our mechanism redirects it through the *gate* library where a decision is made on how it will proceed and if it is allowed to access the information securely stored in the private pages.

IV. Implementation

Based on our design, there are two aspects to our approach: (a) compartmentalization and (b) private memory mapping.

A. Compartmentalization

In order to compartmentalize the running application based on its libraries, we separate all the executable Virtual Memory Areas (VMAs) and map a custom *gate* library in the process's memory space, one for each identified VMA. Aiming to adhere to the library-level granularity of our design (i.e., intercept only calls between libraries and not internal ones), after we make all the VMAs non-executable (NX), we then follow the procedure depicted in Figure 3, when transitioning from one library to another. We first check the previous address where we caused a deliberate fault to determine if it corresponds to the same VMA as the current one (meaning same executable/library) (Figure 3 (1)), in which case we leave the VMA as executable (the current VMA needs to

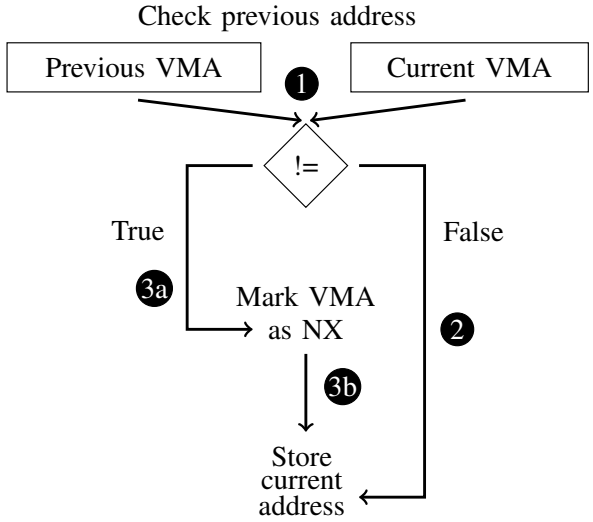


Figure 3. Compartmentilizing an application at library-level granularity

be executable by default, in order not to disrupt execution) and store the current faulting address in a custom field in the process (Figure 3 (2)) .

If the previous and current VMAs are different (meaning different executables/libraries by extension), we mark the previous VMA as NX (Figure 3 (3a)), before storing the current faulting address in the custom field (Figure 3 (3b)). At this stage all the addresses of our process are in a non-executable state, but the execution is able to continue since it is in the context of the Page Fault Exception Handler (PFEH) [28] that intervened to rectify our deliberate page fault, which we caused in order to intercept the call. From there it is redirected inside the *gate*, where a security policy can be applied in order to determine whether to allow the call to access the requested information inside the protected memory and to continue to the originally-intended path. When the PFEH intervenes to rectify the next fault, the same procedure is followed from the top.

B. Private Memory Mapping

Following the separation of the process's memory area into regions, we are now ready to associate private memory pages with each of them. After mapping the *gates*, we introduce protected pages to the process's memory space, where we can save sensitive information that need protection against disclosure, tampering, execution, etc. These pages are only mapped when the CPU executes code within the associated library. When execution is transferred outside the library, the pages get unmapped, thus protecting data stored in them from unauthorized access.

This whole procedure is performed automatically on the kernel side, without requiring access to the source code/binary of the application or linked libraries, thus making our approach completely transparent.

 ISBN: 978-1-61208-919-5

Application Programming Interface: In order to facilitate the use of this extended capability, we propose an Application Programming Interface (API) analogous to the one used for shared memory [29]. Listing 1 showcases a sample of our proposed API, where the code has the ability to allocate a private memory space to a specific region.

```
1  ...
2  char *addr;
3  int fd;
4  fd = scrm_open(PAGE_SIZE, <FLAGS>);
5  addr = mmap(NULL, PAGE_SIZE,
6              PROT_READ | PROT_WRITE,
7              MAP_PRIVATE, fd, 0);
8  scrm_assoc(<caller>, fd, addr,
9             addr + PAGE_SIZE);
10 ...
11 scrm_unlink(fd);
12 ...
```

Listing 1. Usage example of Secure API

First, we create a secure memory (scrm) object with specific flags and its size set to that of a page (line 4). Then we map the object into the process's address space (line 5). Following, we associate the object with the caller (a given region) in line 8. Finally, after some processing we return the memory to the system, by unlinking the scrm object.

V. Evaluation

In order to evaluate the performance overhead incurred by our mechanism, we use the OpenSSL benchmark test of Phoronix Test Suite (PTS). Our test-bed can be seen in Figure 4, as reported by PTS.

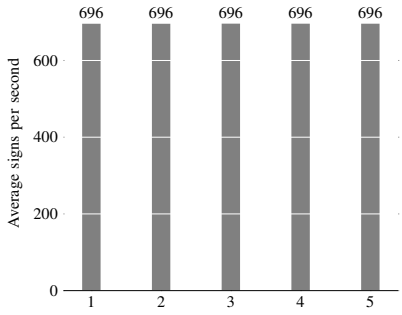
PHORONIX-TEST-SUITE.COM	Phoronix Test Suite 9.0.1
AMD FX-8370 Eight-Core @ 4.00GHz (4 Cores / 8 Threads)	Processor
ASRock 970M Pro3 (P1.60 BIOS)	Motherboard
AMD RD9x0/RX980	Chipset
16384MB	Memory
256GB SAMSUNG MZ7TD256	Disk
Sapphire AMD Radeon HD 6450/7450/8450 / R5 230 OEM 1GB	Graphics
Realtek ALC892	Audio
DELL U2412M	Monitor
Realtek RTL8111/8168/8411	Network
Ubuntu 16.04	OS
4.16.7.CUSTOM (x86_64)	Kernel
Unity 7.4.5	Desktop
X Server 1.19.6	Display Server
modesetting 1.19.6	Display Driver
3.3 Mesa 18.0.5 (LLVM 6.0.0)	OpenGL
GCC 5.4.0 20160609	Compiler
ext4	File-System
1920x1200	Screen Resolution

Figure 4. System configuration

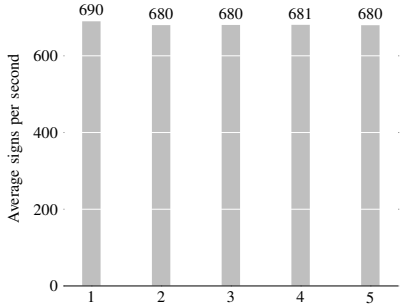
PTS [30] is an open-source automated benchmarking suite that supports a variety of platforms, including Linux. We use it to run a benchmark test for the OpenSSL library, which is executed five times, for both cases: (a) default MMU, (b) MMU customized with our mechanism. The outcome reports on the performance of RSA 4096-bit signing. Figure 5 summarizes the results of the tests (rounded numbers), while Table I shows the deviation for each run. As is evident, there is only minimal decrease in performance - about 2% on average - when using our custom MMU, which makes our approach very efficient.

TABLE I
Deviation for each run of the OpenSSL benchmark of PTS

	Deviation	
#	Default MMU	Custom MMU
1	0.02%	0.04%
2	0.08%	1.3%
3	0.16%	1.23%
4	0.02%	1.32%
5	0.16%	1.32%



(a) Default kernel



(b) Custom kernel

Figure 5. Performance evaluation of our mechanism using the OpenSSL benchmark of PTS

A. Memory Coverage Analysis

In order to measure to what degree our mechanism compartmentalizes a program's memory space and by extension confines an attacker's code base that is available at any given point in time for them to mount an attack, we analyze four well-known applications, i.e., NGINX HTTP server, VMware Player, Sublime Text Editor and GNOME MPlayer - with respect to their executable memory areas. We chose these applications for analysis, based on their broad acceptance and usage in their respective domains in a Linux environment.

In Table II, we can see the result of the analysis for the NGINX HTTP server. We only measure the size of the

TABLE II
MEMORY COVERAGE OF LIBRARIES FOR THE NGINX MAIN APPLICATION

Library	Size (in bytes)	% of total
nginx (main)	528384	6.84%
libnss_files	45056	0.58%
libnss_nis	45056	0.58%
libnsl	90112	1.17%
libnss_compat	32768	0.42%
libdl	2093056	27.11%
libc	1835008	23.77%
libz	102400	1.33%
libcrypto	2207744	28.59%
libpcre	450560	5.84%
libcrypt	36864	0.48%
libpthread	98304	1.27%
ld	155648	2.02%
Total	7720960	100%

TABLE III
LIBRARIES WITH MAXIMUM COVERAGE FOR THE OTHER THREE APPLICATIONS

Application	Library	% of total
VMware Player	libvmwareui	21.53%
Sublime Text Editor	libgtk-3	20.63%
GNOME MPlayer	libicudata	34.74%

executable VMAs, since all others are out of scope. We can see that the biggest memory area corresponds to *libcrypto* and it takes up around 29% of the program's total executable memory. Similarly, based on our analysis of the other three applications (Table III) - details of which we omit for the sake of space, since they are composed of tens of libraries - we can see that at the maximum only a small portion of the address space is available to the attacker at any given moment, which results in them having much lower chances of success when trying to launch a CRA. If we also consider that each of these smaller regions has one or at most a few pages of memory dynamically associated with it, where only it has access and can store sensitive code and data, it becomes even clearer that a rogue (part of an) application will find it extremely difficult to gain access to this information and compromise the system.

B. Real-life Scenarios

In this section, we present two examples that showcase the applicability of our defense mechanism.

First, let's consider the case of the Dual Elliptic Curve Deterministic Random Bit Generator (Dual_EC_DRBG) backdoor. Dual_EC_DRBG [31] was presented as a cryptographically-secure pseudorandom number generator that used elliptic curve cryptography. Despite the fact that there were several weaknesses publicly identified, one of which being a backdoor that could only be exploited by someone who knew about it (presumably the United States government's National Security Agency), the algorithm was adopted as a standard by several standardization bodies. In such a case, using our mechanism we would not have to wait for a patch/updated version to be released or some other kind of action to be taken by the responsible parties (later the algorithm was withdrawn). Upon detecting a call to one of the Dual_EC_DRBG-related functions, we immediately produce a warning/error informing that this specific generator contains vulnerabilities, and/or prevent the call to continue to the intended function (we can also disable/remove the algorithm from the results when reporting which pseudorandom number generators are available in a library). In this way, our defense acts more as a preventive measure and less as a responsive one after the fact, protecting the user even before an attacker gets a chance to exploit the vulnerabilities. Even in the case of such a widely-adopted algorithm, used by a number of official bodies, our approach would be able to offer sufficient information to the users to make an informed decision.

The second scenario deals with handling a private key. In this case, we leverage the OpenSSL library and specifically its *libcrypto*/*libssl* libraries. When a program needs to sign a piece of data (text, file, etc.), it needs access to a private key. Under our scheme, when a call to a function from these libraries is intercepted, it is redirected inside the *gate*, where we forbid it to access the private key directly. We have already included a secure function in the *gate* - `sec_pkey()`, which is the only one that can access the secure private memory associated with OpenSSL, where the key is stored. There is a number of ways the key can be placed in memory: (a) after the program starts, we read the private key from a file with elevated privileges, store it in private memory and then close the file. From then on we revoke access to the file, which means that access to the key is provided only through the *gate* and OpenSSL's private memory, (b) the program creates its own private key and places it in memory, or (c) the key is initially retrieved from a file and stored in memory *lazily*, i.e., only when there is a call to an OpenSSL function.

`sec_pkey()` retrieves the key, signs the data and returns the result. This way, the rest of the program does not have access to the private key. Inside `sec_pkey()` we can perform a number of checks to verify that only a specific legitimate OpenSSL function requested access to the key and that was only to read it and at an appropriate point in execution time. To determine at which point the execution is, we can store in private memory a finite state automaton/state model of the application, which e.g., we have created by running the application through our custom MMU in learning mode or the developer has provided us with. Inside the *gate*, we also have a relevant function that is responsible for checking the program state `chk_stt()`, that checks several parameters (e.g., depth/size of stack, call origin/destination, number of call parameters, etc.) and their combinations to determine if the current state corresponds to the one saved in the model. This way, we can make sure that execution is at the correct point in time and that nothing has interfered with the execution flow.

VI. CONCLUSION

In this paper, we present an extension of our previous work in [2] [3] where, after separating the memory of a running

process into regions at the granularity of executables/external libraries, it maps private pages for each region that are only accessible from the associated *gate*, leveraging the MMU. Our approach is very efficient and transparent and can be used on binary/legacy applications and existing environments, as well as serve as a complimentary measure of defense alongside already implemented mechanisms. Furthermore, we present two scenarios where our mechanism can protect real-life applications.

ACKNOWLEDGMENTS

This work is supported by the European Commission through the following H2020 projects: SENTINEL under Grant Agreement No. 101021659, ROXANNE under Grant Agreement No. 833635 and CONCORDIA under Grant Agreement No. 830927.

REFERENCES

[1] M. Tsantekidis and V. Prevelakis, "Library-Level Policy Enforcement," in *SECURWARE 2017, The Eleventh International Conference on Emerging Security Information, Systems and Technologies*, Rome, Italy, 2017, [Retrieved: 10-2021]. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=securware_2017_2_20_30034

[2] M. Tsantekidis and V. Prevelakis, "Efficient Monitoring of Library Call Invocation," in *Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, Granada, Spain, 2019, [Retrieved: 10-2021]. [Online]. Available: https://doi.org/10.1109/IOTSMS48152.2019.8939203

[3] M. Tsantekidis and V. Prevelakis, "MMU-based Access Control for Libraries," in *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT*, INSTICC. SciTePress, 2021, pp. 686–691, [Retrieved: 10-2021]. [Online]. Available: https://www.scitepress.org/Link.aspx?doi=10.5220/0010536706860691

[4] S. T. Alliance, "Trusted Execution Environment (TEE) 101: A Primer," 2018, [Retrieved: 10-2021]. [Online]. Available: https://www.securetechalliance.org/wp-content/uploads/TEE-101-White-Paper-FINAL2-April-2018.pdf

[5] N. Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "Mobile trusted computing," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1189–1206, 2014.

[6] L. Limited, "Open Portable Trusted Execution Environment," 2021, [Retrieved: 10-2021]. [Online]. Available: https://www.op-tee.org/

[7] GlobalPlatform, "Trusted Execution Environment (TEE) Committee," 2021, [Retrieved: 10-2021]. [Online]. Available: https://globalplatform.org/technical-committees/trusted-execution-environment-tee-committee/

[8] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 559–572.

[10] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 30–40.

[12] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 574–588.

[13] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 227–242.

[14] J. Habibi, A. Panicker, A. Gupta, and E. Bertino, *DisARM: Mitigating Buffer Overflow Attacks on Embedded Devices*. Cham: Springer International Publishing, 2015, pp. 112–129.

[15] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli, "A high-performance, low-overhead microarchitecture for secure program execution," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Sept 2012, pp. 102–107.

[16] M. Graziano, L. Flore, A. Lanzi, and D. Balzarotti, *Subverting Operating System Properties Through Evolutionary DKOM Attacks*. Cham: Springer International Publishing, 2016, pp. 3–24.

[17] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. A. Ghazaleh, "Signature-based protection from code reuse attacks," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 533–546, Feb 2015.

[18] M. Hoekstra, "Intel® SGX for Dummies (Intel® SGX Design Objectives)," 2015, [Retrieved: 10-2021]. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx

[19] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10327 LNCS. Springer-Verlag Italia, 2017, pp. 3–24.

[20] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 217–233.

[21] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A. Sadeghi, "DR.SGX: hardening SGX enclaves against cache attacks with data location randomization," *CoRR*, vol. abs/1709.09917, 2017.

[22] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy*, 2019.

[23] D. O'Keeffe, D. Muthukumaran, P.-L. Aublin, F. Kelbert, C. Priebe, J. Lind, H. Zhu, and P. Pietzuch, "Spectre attack against SGX enclave," 2015, [Retrieved: 10-2021]. [Online]. Available: https://github.com/lsds/spectre-attack-sgx

[24] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[25] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 523–539.

[26] M. Schwarz, S. Weiser, and D. Gruß, "Practical enclave malware with Intel SGX," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science. Springer International, 2019, pp. 177–196.

[27] S. Weiser, L. Mayr, M. Schwarz, and D. Gruss, "SGXJail: Defeating enclave malware via confinement," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Sep. 2019, pp. 353–366.

[28] D. P. Bovet and M. Cesati, "Page Fault Exception Handler," in *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2005, ch. 9.4.

[29] M. Kerrisk, "shm_overview(7) — Linux manual page," 2008, [Retrieved: 10-2021]. [Online]. Available: https://www.man7.org/linux/man-pages/man7/shm_overview.7.html

[30] PTS, "Phoronix Test Suite," [Retrieved: 10-2021]. [Online]. Available: https://www.phoronix-test-suite.com

[31] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," National Institute of Standards and Technology (NIST), Tech. Rep., 2012, [Retrieved: 10-2021]. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-90a.pdf

 ISBN: 978-1-61208-919-5