Software Side-Channel Attacks Against Intel SGX Enclaves

CS 658 Research Paper

Colin Howes
University of Waterloo
200 University Ave. W
Waterloo, ON N2L 3G1
chowes@uwaterloo.ca

ABSTRACT

Intel Software Guard Extensions (SGX) is a hardware-backed security extension of the Intel architecture that allows user-level software to run securely in an environment where all other software on the host system is untrusted. SGX uses secure enclaves running in Processor Reserved Memory combined with a software attestation scheme to provide confidentiality and integrity guarantees to users wishing to execute software on an untrusted remote system. SGX is vulnerable to a number of software side-channel attacks, which leverage performance measurements to determine memory access patterns and derive secrets from software executing in secure SGX enclaves. SGX vulnerabilities to software side-channel attacks, possible countermeasures, and directions for future research are discussed.

1 BACKGROUND

Users have traditionally relied on services offered by the operating system to provide security and privacy protections by exerting strict control over resource management and enforcing logical isolation of user-level processes. In a cloud computing scenario, users wishing to run software on remote systems may not trust the cloud service provider responsible for administering the host operating system or hypervisor. It is therefore necessary to consider a scenario where a user wishes to securely execute his or her software on a system controlled by a potentially malicious service provider with control over privileged software. Intel Software Guard Extensions aims to address this problem by providing a Trusted Execution Environment through the addition of hardware extensions that provide strong confidentiality and integrity guarantees in an environment where no software on a host machine is trusted. This section summarizes the general functionality of Intel Software Guard Extensions, and provides a background on software side-channel attacks.

1.1 Intel SGX

Intel Software Guard Extensions (SGX) is an extension to the Intel architecture designed to allow programs to run securely in an environment where all software on the host machine is potentially untrusted [3]. SGX was designed to address the problem of secure remote computation, that is, secure execution of software on a remote system controlled by an untrusted party [3]. Under this threat model, *all* software on a remote system is potentially malicious, including privileged software such as the operating system (OS) and hypervisor. Thus SGX must provide protection to user-level programs from untrusted software running at user level as well as at higher privilege levels [4].

SGX provides confidentiality and integrity guarantees through the use of trusted hardware that sequesters a user-level program into a secure container called an *enclave*, and then proves to a user through *software attestation* that he or she is running unmodified software protected in an enclave by secure hardware [3]. In order to provide software attestation, the host machine provides a cryptographic signature certifying an enclave's measurement hash, which is computed on the contents of an enclave after all necessary data has been loaded from unprotected memory into enclave pages secured in Processor Reserved Memory, and an initialization process has been completed [4]. The remote user can then verify this signature against an endorsement certificate provided by the hardware manufacturer, and can refuse to load his or her data into an enclave whose measurement hash does not match the expected value [3, 4]. Under this threat model, the remote user only needs to trust the hardware manufacturer responsible for providing the secure hardware and endorsement certificate.

1.1.1 SGX Enclaves. The SGX security model is centered around maintaining security-sensitive information in secure containers isolated from the rest of the untrusted host system. SGX accomplishes this through the use of a contiguous range of protected memory called Processor Reserved Memory, which is accessible only via a set of specialized microcode instructions called from within an enclave [4]. Enclave contents and meta-data are stored in a subset of Processor Reserved Memory called the Enclave Page Cache, which is split into 4 KB pages. Management of enclave pages is delegated to the untrusted host OS or hypervisor, which can manage pages via a limited interface using SGX microcode instructions, but may not access this memory directly. Rather, enclave pages are encrypted while in DRAM, and are decrypted in hardware as they are loaded into the cache by the CPU. Furthermore, memory management decisions made by system software are tracked in order to maintain isolation of protected memory by verifying that enclave pages can only be accessed by enclave code executing in the enclave associated with a given enclave page [3, 4, 10].

1.2 Software Side-Channel Attacks

Side-channel attacks are a class of attacks that leverage information about the physical properties of a system in order to carry out an attack. Physical side-channel attacks use knowledge of the characteristics of a system to infer secrets from physical information leaks. For example, an attacker can analyze power consumption or electromagnetic radiation to derive secrets from a machine executing otherwise secure computations [15].

Software side-channel attacks exploit information leaks that result from characteristics of the hardware or software implementations of a secure system that can be observed by software running on the system itself. For example, an attacker can take advantage of the difference in time needed to access data stored in the cache compared with data stored in main memory to infer secrets from memory usage patterns [4].

While physical side-channel attacks against modern hardware are difficult and costly, requiring advanced tools and physical access to the victim machine, software side-channel attacks are inexpensive to deploy and can be executed by anyone with remote or local access to a system [4]. The software side-channel attacks discussed here exploit hardware and software implementation details to acquire information about memory access patterns, which can be used to infer secrets from an otherwise secure system [7, 11, 14, 16]. Side-channel attacks are not included in the SGX threat model due to the perceived impracticality of this class of attacks [3, 4]. However, as the work reviewed below demonstrates, software side-channel attacks are practical against SGX enclaves and pose a significant threat to program security in a cloud environment where privileged software cannot be trusted [2, 10].

2 ATTACKS AGAINST SGX ENCLAVES

A number of distinct software side-channel attacks have been demonstrated against SGX enclaves in proof-of-concept attacks. While enclave data is protected in Processor Reserved Memory and is isolated from both OS and user-level processes, memory management and scheduling is deferred to the untrusted OS and/or hypervisor, giving a malicious OS a great degree of control over a victim SGX process and providing fine grained measurement capabilities. Thus, a malicious OS is free to dictate precisely how and when a victim enclave program is scheduled, and on which logical cores. This high magnitude of control over the victim process can be leveraged to carry out powerful privileged software side-channel attacks against SGX enclaves [4].

Moreover, the protection and isolation guarantees provided by SGX extend to malicious programs, meaning that an adversary can use an SGX enclave to launch attacks against a victim enclave running on the same host system. Since even privileged services are oblivious to the contents and behavior of SGX enclaves, the host OS has no means to monitor its SGX enclaves for malicious behavior, and thus cannot protect honest user-level processes from malicious enclave behavior [4, 11].

2.1 Cache Timing Attacks

Caches are small regions of high-speed memory used to store recently accessed data in order to reduce future access times. In modern computers with multiple cache levels, each core typically has its own first and second (L1 and L2) caches, with a third level (L3) cache, also referred to as the lowest level cache (LLC), shared across all cores. When the CPU needs to access data, accesses may be resolved in a cache (a cache "hit"), or data may need to be fetched from main memory (a cache "miss"), at which point it is stored in the cache, evicting older data. Cache accesses are considerably faster than memory accesses that must be resolved in main memory, and so caching data can significantly reduce computation time by reducing reliance on relatively slow main memory accesses. Cache timing attacks exploit the timing differences between cache hits and misses to infer secrets kept by a concurrently executing program by analyzing the victim processes memory access patterns [4, 10].

At a high level, an attacker can take advantage of this difference in access speed by filling a shared cache with data, thereby evicting data used by a victim process. The attacker then measures the time it takes to access this data after the victim has a chance to run. Blocks of memory that take longer to access were likely evicted from the cache to make space for data used by the victim process, implying that the victim process accessed memory mapped to the same region of the shared cache. Since blocks of memory are mapped to regions of cache storage based on virtual memory address, the attacker can use a cache attack to derive information about the victim's memory usage patterns, which can be used to extract secrets such as encryption keys [9]. In this way, the attacker can infer which regions of memory were accessed by the victim based on the time needed to access the data it had previously placed in the cache [7, 10].

The Prime+Probe attack is a form of cache attack that can be used to attack a victim process by targeting a shared cache. Usually, this attack is used to target the L1 cache on a multithreaded core shared between the attacker process and a victim process, though it can also be adapted for use in the LLC. In a *priming* step, the attacker fills the entire shared cache, thereby causing the victim's cached data to be evicted. Next, the attacker waits for the victim process to run, at which point it makes memory accesses that result in attacker memory blocks being evicted from the shared cache. Finally, in a *probe* step, the attacker attempts to access its data, timing memory accesses to determine which blocks of memory were used by the victim process [7, 10].

Proof of concept attacks against a number of AES implementations have demonstrated that an attacker can launch powerful Prime+Probe cache attacks against SGX enclaves by taking advantage of OS-level scheduling privileges and performance monitoring tools. By leveraging control over a malicious host OS to schedule a Prime+Probe attack program

on the same core as the victim SGX enclave with all other software isolated to other cores, an attacker can eliminate noise caused by the execution of other processes changing the cache set. In addition, since the attacker can also control scheduling, the attacker process can frequently interrupt the victim, providing very good temporal resolution - that is, by reducing the time that the victim is allowed to execute before the attacker can probe for cache misses, the attacker can make stronger inferences about the timing and order of memory accesses by the victim process. Thus, an attacker with control over a malicious OS can leverage control over scheduling decisions to carry out powerful high-resolution cache attacks capable of deriving secret keys from victim enclaves executing cryptographic routines without needing to directly circumvent the protections offered by SGX [7, 10].

Moreover, a remote attacker can take advantage of the protections offered by SGX enclaves to conceal malicious code and carry out cache attacks against co-located victim enclaves using only resources available to user-level processes. In this case, the attacker has no control over where or when the attacker and victim processes will be scheduled, and so the attacker cannot take advantage of timing or scheduling decisions to improve the noise or resolution of the leaked information. However, since the OS has no straightforward means of monitoring the behavior of enclaves hosted on the system, side channel attacks carried out from malicious enclaves can be concealed from antivirus software by the protections offered by SGX. Such an attack has been shown to be capable of extracting private keys from an enclave running a current RSA implementation [11]. This demonstrates that SGX enclaves are not only vulnerable to attacks launched by a privileged adversary leveraging the power of a malicious OS, but also to attacks carried out by remote adversaries who can use the confidentiality guarantees offered by SGX to make themselves indistinguishable from honest users.

2.2 Page-Fault Attacks

Similar to cache timing attacks, *controlled-channel* page-fault attacks allow an attacker to derive secrets from a victim program based on memory access patterns. However, unlike cache timing attacks, controlled-channel page-fault attacks rely on an attacker leveraging control over system events to force a victim process to generate page-faults, directly revealing information about its memory access patterns that can be used to infer secrets [16].

Controlled-channel page-fault attacks rely on an attacker having control over memory management, as is the case when an attacker controls a malicious OS running on a system upon which a victim is executing code in an SGX enclave [2]. In order to carry out an attack, the attacker must first analyze the victim's source code or binary executable in order to identify input-dependent data accesses and input-dependent control transfers. When the application is run, the attacker can restrict access to a target set of the victim's pages, causing a page-fault to be triggered whenever the victim attempts to access an address on a target page. Since the CPU must report at least the base address of the faulting page to the OS, the attacker can record sequences of page accesses and can use this information to run an offline analysis and make inferences about the victim's secret data at page level (4 KB) granularity [14, 16].

Controlled-channel page-fault attacks are feasible against SGX enclaves since a privileged attacker has control over memory management, and SGX aims to support legacy applications without requiring that they be modified to conceal private data from an attacker that can observe memory access patterns [16]. Moreover, a malicious OS can exert fine-grained control over page allocation to carry out a "pigeonhole" attack against an SGX enclave. Here, the OS allocates only 3 pages to the victim enclave - a code page, a source page, and a destination page - which make up the "pigeonhole set". Any memory accesses that fall outside of these pages will cause a page-fault, revealing to the OS which data or code the application is attempting to access. By repeatedly causing the victim to generate page-faults and adjusting the pigeonhole set, the attacker can carefully observe the sequence of page accesses made by the victim, maximizing the information leaked to the attacker via the page-fault side-channel. This technique has been demonstrated against SGX enclaves running implementations of OpenSSL and Libgcrypt [14].

The operating system's high degree of control over system events enables these powerful, no noise side-channel attacks, which could not be carried out in a traditional computing environment where the operating system is trusted. Though SGX's threat model explicitly excludes side-channel attacks, the practicality of these mechanisms as attack vectors cannot be ignored given that SGX allows the OS to exert complete control over memory management and scheduling.

2.3 Branch Prediction Analysis

Modern processors use instruction *pipelining* to fetch and decode instructions while previous instructions are executing. In order to maintain efficiency when a branch is encountered, the processor predicts which instruction will be executed after the branch based on past behavior. The predicted next instruction is fetched before the processor has determined which branch must be taken, and if the prediction is correct pipelined execution can continue as normal. On a branch misprediction the processor must clear the predicted instruction and load the correct instruction before execution can continue. In either case, the outcome is used as the basis for prediction the next time the branch is encountered [1, 8].

An attacker can leverage this behavior to infer information about a victim process through a *branch shadowing attack*. Briefly, an attacker uses knowledge of a program's structure to construct a "shadow" program, which has branches at the same points in execution as the victim. The attacker can then interrupt the victim process after a branch is encountered and run the shadow program starting at the address of the branch most recently encountered in the victim program. The attacker can then use timing differences or performance

monitoring tools to determine whether the branch predictions made by the processor based on the behavior of the victim led to correct or incorrect predictions of the shadow program's behavior, and can infer information about the victim's execution patterns [8].

SGX does not clear an enclave's branch history when performing an enclave exit, and a privileged attacker can take advantage of this by forcing frequent interrupts such that all branches taken by the victim can be determined by the attacker. In order to reduce the noise inherent in attempting to time differences in performance between correct and incorrect branch predictions in the shadow program, an attacker can make use of the Last Branch Record (LBR), a CPU debugging feature that records the last branch taken and whether the CPU's prediction was correct. This feature is only available in debugging mode, so the attacker must set the shadow program to run in debugging mode and check whether the predictions based on the behavior of the victim led to correct choices in the shadow program. The attacker can therefore use this privileged debugging mechanism to determine precisely which branches were taken by the victim and infer information about the input to the victim program. Proof of concept attacks have demonstrated that this technique can reveal information about input and program behavior protected in SGX enclaves that cannot be detected by the controlled-channel page-fault attacks described above [8, 14]. Here, an attacker can make use of debugging mechanisms made available by hardware to privileged software to eliminate noise in an otherwise impractical side-channel.

3 COUNTERMEASURES

Side-channel attacks against SGX enclaves can take advantage of privileged control over memory management and scheduling as well as access to performance monitoring and debugging mechanisms to infer secrets from SGX enclaves. In addition, user-level attacks can use the protections afforded by SGX enclaves to conceal attacks against other enclaves running on the same system. A number of countermeasures have been proposed that aim to prevent sensitive information from leaking over the side channels discussed above, including software- and compiler-based strategies that can be implemented to harden software running on current SGX implementations, as well as hardware-based solutions seeking to expand the guarantees provided by hardware-backed Trusted Execution Environments.

3.1 Software and Compiler Countermeasures

Software- and compiler-based prevention mechanisms are particularly attractive since these defenses allow developers to protect current SGX applications from side-channel attacks with minimal code modification. In addition, these techniques can be implemented immediately, meaning that users wishing to take advantage of SGX can do so without needing to wait for Intel or its competitors to implement an effective hardware solution addressing the side-channel vulnerabilities discussed above.

3.1.1 SGX-Shield. SGX-Shield is a software-based protection mechanism that provides Address Space Layout Randomization (ASLR) to SGX programs. SGX-Shield is itself an enclave program comprised of a multistage loader that loads an enclave into randomized addresses, ensuring that the untrusted OS is oblivious to the address space layout of the protected program. The main goal of SGX-Shield is to protect enclaves against memory corruption attacks, though SGX-Shield also provides protection against cache timing and page-fault attacks as a side effect. Since SGX-Shield randomizes the order in which data is loaded and relocated, input-dependent memory accesses must be determined by a malicious kernel at run-time via brute-force guessing, which is likely to cause a crash in the case of a controlled-channel page-fault attack [12].

3.1.2 Deterministic Multiplexing. This proposed compiler-based technique for preventing page-fault attacks against secure enclaves strives for *page-fault obliviousness* by making page accesses the same for all inputs. This approach is based on the premise that page-fault attacks infer secrets by observing page-faults that leak information about input-dependent data accesses and input-dependent control transfers. In order to achieve this, a balanced execution tree is constructed by inserting "dummy" execution blocks, and all code and dummy blocks for a given "level" of execution are placed into the same page. Thus, the same page is loaded into memory regardless of which branch in execution is taken, preventing the OS from observing input-dependent control transfers. Similarly, all data required by branches at a given level of execution are also placed into the same page, thereby making it impossible to determine for the OS to observe input-dependent data accesses by observing memory access patterns at page level granularity [14].

Thus, an attacker sees page-faults implying that any of the possible execution paths could have been taken by the victim, and cannot make inferences about input based on data accesses since all relevant data is on the same page in memory. This *deterministic multiplexing* is carried out by the compiler based on annotations made by the programmer, and makes it impossible for an attacker to infer secrets based memory access patterns since page accesses will be identical for all inputs [14]. This technique is not effective against other attacks such as cache timing attacks or branch shadowing, so deterministic multiplexing alone is not sufficient to protect SGX enclaves against all classes of side-channel attacks [8, 14]. In addition, the overhead associated with creating dummy execution blocks is non-negligible, meaning that programs with many branches or deep execution trees incur massive overhead [14].

3.1.3 T-SGX. T-SGX is a compiler-based approach to protect against controlled-channel page-fault attacks through the novel use of Transactional Synchronization Extensions (TSX), an additional extension available on modern Intel

processors designed to implement "hardware transactional memory" as an alternative to traditional software-based synchronization primitives. Briefly, TSX allows a process to initiate "transactional execution", which stores the results of short "transactions" in the L1 cache, writing the results into memory only if it is certain that no conflicts have occurred [13].

In the case of T-SGX, any interrupted transaction is invalidated, effectively allowing exceptions such as page-faults to be routed to a TSX abort handler protected inside the enclave. T-SGX therefore allows an enclave to detect suspicious exceptions and abort execution if the OS has unmapped any of its sensitive memory pages. In effect, T-SGX introduces compiler mechanisms that transform SGX enclave programs into series of small uninterruptible single-page transactions. These small execution blocks are launched by a "springboard" page, which also manages the TSX abort handler. Since this springboard page is the only code executed outside of a transaction, a malicious OS can only force malicious page-faults on the springboard page, which cannot be used to infer sensitive information. Through these mechanisms, T-SGX ensures that the OS cannot arbitrarily evict enclave pages from memory in order to derive secrets from memory access patterns, and aborts execution if it detects malicious manipulation of enclave pages by the untrusted OS [13]. Thus, T-SGX obviates controlled-channel page-fault attacks by ensuring that sensitive computations cannot be interrupted by the OS, and by detecting suspicious page-faults and terminating execution if necessary.

Additionally, T-SGX allows an enclave program to flush its private cache whenever it resumes execution, which is sufficient to prevent the Prime+Probe cache attacks targeting the L1 cache discussed above, but is not enough to prevent cache timing attacks targeting the LLC [13]. Furthermore, T-SGX enclaves remain vulnerable to branch shadowing attacks since an attacker can use branch shadowing to determine the order in which the springboard launches its execution blocks without triggering the T-SGX abort handler [8].

3.2 Hardware Extensions

The goal of SGX is to provide a secure computing solution to users wishing to execute legacy code on remote machines in an untrusted cloud environment without needing to implement special protections directly or make major changes to existing software. To this end, future iterations of SGX and its competitors will likely attempt to address software side-channel vulnerabilities through hardware modifications and changes to the SGX microcode in order to avoid the performance overhead and complexity associated with software- and compiler-based mitigation strategies.

3.2.1 Sanctum. Sanctum is a hardware-based extension to the Rocket RISC-V core design providing the same guarantees as SGX along with additional protections against software side-channel attacks. Sanctum's design is similar to that of SGX but diverges in its use of a software-based security monitor that runs at a higher privilege level than the OS or hypervisor, whereas SGX is implemented almost entirely in microcode [3–6].

Sanctum's security monitor is able provide stronger protection against cache-timing attacks to enclave programs than SGX by directing interrupts to the security monitor rather than directly to the untrusted OS. The security monitor flushes the TLB and per-core caches before initiating an enclave exit and allowing the OS to handle an interrupt or fault, thus preventing a malicious OS from carrying out cache-timing attacks against per-core caches. In addition, the security monitor shifts the physical address of enclave data before it is stored in the shared LLC, so a malicious OS wishing to carry out a cache-timing attack against the shared LLC has no way of determining which cache set stores its target memory address [5, 6].

In addition, Sanctum provides protection against page-fault attacks by assigning private page tables to enclaves and having enclaves handle their own page-faults, while allowing the OS to manage memory management by assigning pages to enclaves [5, 6]. In SGX, page table management leaks information to the untrusted OS, allowing the OS to make inferences based on an enclaves memory access patterns [4–6].

Nevertheless, the role of the OS in providing an abstracted view of memory to concurrently executing programs is critical to the overall efficiency of modern systems, and removing paging or OS-controlled memory management is not a feasible solution to the controlled-channel vulnerabilities in SGX. Thus, Sanctum delegates course-grained memory management to the OS, which is responsible for allocating and deallocating regions of memory to enclaves. In order to prevent information from leaking through the page-fault channel, private page tables are assigned to each enclave, and enclave page-faults are serviced by a handler private to each enclave and managed by the security monitor [5, 6]. In this way, the role of the OS in managing memory resources is preserved while eliminating information leaks via the page-fault channel.

3.2.2 Hardware Protections against Branch Shadowing. Despite its guarantees of isolation and protection against privileged attackers, Sanctum appears to be vulnerable to branch shadowing attacks since these attacks do not rely on an attacker generating page-faults or determining the physical addresses that correspond with target addresses in virtual memory. Fortunately, preventing branch shadowing attacks is relatively simple to accomplish at the hardware level. In the context of Intel's architecture, preventing branch shadowing attacks requires that branch prediction data structures be isolated to a given hardware thread in order to prevent attacks that take advantage of hyperthreading, or that hyperthreading be disabled altogether. In addition, branch prediction data must be flushed on every enclave exit in order to prevent a shadowed program from making inferences about an enclaves branch history [5, 6, 8].

4 FUTURE DIRECTIONS

Since the SGX threat model affords an attacker with such a wide attack surface, future research should continue to

search for channels through which sensitive information can leak from an enclave and taken advantage of by an attacker with control over a malicious OS or hypervisor. In addition, most countermeasures have focused on mitigating attacks carried out by a privileged attacker with little emphasis on the danger of a malicious user-level enclave using the protections provided by SGX to disguise attacks against other software running on the same host. Finally, future work should continue to search for countermeasures that can either be included in future iterations of SGX and its competitors, or integrated into SGX enclaves in order to prevent or mitigate software side-channel attacks.

5 CONCLUSIONS

Intel SGX enclaves are vulnerable to a number of software side-channels that can be implemented cheaply and easily by an attacker with remote or local access to the host machine. Privileged attackers can take advantage of control over scheduling and memory management, as well as powerful performance monitoring tools, while unprivileged attackers can use the protections afforded by SGX enclaves to disguise attacks directed at other enclaves running on the same host.

A number of software- and compiler-based prevention mechanisms have been proposed, but these techniques involve added overhead and complexity, and none of the techniques discussed here provide full protection against all identified side-channel vulnerabilities. The Sanctum hardware extensions follow the same design principles as SGX and make the same guarantees, but also provide stronger isolation for enclaves and added protection against software side-channel attacks.

In order to effectively provide security guarantees to users running programs in a cloud environment, SGX and other trusted execution environments must take software side-channel attacks into consideration and effectively prevent a privileged attacker from inferring secrets via performance monitoring or debugging mechanisms. The difficulty in providing these assurances lies in the wide attack surface that control over a malicious OS affords an attacker, though a number of software-, compiler-, and hardware-based solutions present promising techniques beyond existing security offered by SGX.

REFERENCES

[1] Onur Aciimez, etin Kaya Ko, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*. ACM, New York, NY, USA, 312–320. https://doi.org/10.1145/1229285.1266999

[2] Intel Corporation. 2015. Tutorial Slides for Intel SGX. (2015). https://software.intel.com/sites/default/files/332680-002.pdf

[3] Intel Corporation. 2016. Intel Software Guard Extensions. (2016). https://software.intel.com/en-us/sgx

[4] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86. https://pdfs.semanticscholar.org/2d7f/3f4ca3fbb15ae04533456e5031e0d0dc845a.pdf

[5] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for String Software Isolation. Austin, TX. http://portal.acm.org/toc.cfm?id=1072530

[6] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. 2015. Sanctum: Minimal RISC Extensions for Isolated Execution. *IACR Cryptology ePrint Archive* 2015 (2015), 564. https://pdfs.semanticscholar.org/7ea7/45e466e988d55981e84dd647adb876b3352e.pdf

[7] Johannes Gtzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Mller. 2017. Cache Attacks on Intel SGX. ACM Press, 1–6. https://doi.org/10.1145/3065913.3065915

[8] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. *n Proceedings of the 26th USENIX Security Symposium* (2017).

[9] Wei Liu, Ariel Di Segni, Ye Ding, and Teng Zhang. 2013. Cache-timing Attacks on AES. *New York University* (2013). https://nyuad.nyu.edu/content/dam/nyuad/departments/research/images/center-institutes-grants/moma-lab/moma-project-cache-timing-attacks-on-aes.pdf

[10] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *arXiv:1703.06986 [cs]* (March 2017). http://arxiv.org/abs/1703.06986 arXiv: 1703.06986.

[11] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clmentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *arXiv:1702.08719 [cs]* (Feb. 2017). http://arxiv.org/abs/1702.08719 arXiv: 1702.08719.

[12] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. http://cps.kaist.ac.kr/papers/ndss17-sgxshield.pdf

[13] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. Internet Society. https://doi.org/10.14722/ndss.2017.23193

[14] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2015. Preventing Your Faults From Telling Your Secrets: Defenses Against Pigeonhole Attacks. *arXiv:1506.04832 [cs]* (June 2015). http://arxiv.org/abs/1506.04832 arXiv: 1506.04832.

[15] Franois-Xavier Standaert. 2010. Introduction to Side-Channel Attacks. In *Secure Integrated Circuits and Systems*. Springer, Boston, MA, 27–42. https://link.springer.com/chapter/10.1007/978-0-387-71829-3_2 DOI: 10.1007/978-0-387-71829-3_2.

[16] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656. https://doi.org/10.1109/SP.2015.45