Quarkslab's website

SOCIAL

- atom feed
- twitter
- github

CATEGORIES

- Android
- Android, ReverseEngineering
- Challenge
- Cryptography
- Development
- Exploitation
- Fuzzing
- Hardware
- Hardware, ReverseEngineering
- Kernel Debugging
- Life at Quarkslab
- Maths
- Obfuscation
- PenTest
- Program Analysis
- Programming
- ReverseEngineering
- Software
- Vulnerability

TAGS

Overview of Intel SGX - Part 2, SGX Externals

Date Thu 02 August 2018 By Alexandre Adamski Category ReverseEngineering. Tags Trusted Execution Environment Intel SGX

This blog post provides the reader with an overview of the Intel SGX technology, as a follow-up to SGX Internals. In this second part, we quickly explain how an application interacts with its enclave. We also detail what pieces of software are included within the SDK and PSW. Finally, we summarize the known attacks and concerns with this technology, as well as conclude on the subject.

Interactions

Conceptually, an SGX enclave can be seen as a black-box that is capable of executing any arbitrary algorithm. This black-box can communicate with the outside world using three different ways presented below.

Enclave Calls (ECALLs)

The application can invoke a pre-defined function inside the enclave, passing input parameters and pointers to shared memory within the application. Those invocations from the application to the enclave are called `ECALL`.

Outside Calls (OCALLs)

When an enclave executes, it can perform an `OCALL` to a pre-defined function in the application. Contrary to an `ECALL`, an `OCALL` cannot share enclave memory with the application, so it must copy the parameters into the application memory before the `OCALL`.

Asynchronous EXit (AEX)

The execution can also exit an enclave because of an interruption or an exception. These enclave exit events are called *Asynchronous Exit Events* (AEX). They can transfer control from the enclave to the application from arbitrary points inside the enclave.

Programming

Trusted Code Base (TCB)

Developing an application that uses an SGX enclave requires to identify the resources to protect, the data structure containing those resources, and the code that manages them. Then, everything that has been identified must be placed inside the enclave. An enclave file is a library that is compatible with the traditional operating systems loaders. It contains the code and data of the enclave, which is in plain text on the disk.

Interface Functions

The interface between the application and its enclave must be designed carefully. An enclave declares which functions can be called by the application, and which functions from the application it can call. Enclave input parameters can be observed and modified by the non-secure code, so they must be checked extensively. As an enclave cannot directly access the services of the OS, it must call its application. Those calls should not expose any confidential information and also are not guaranteed to be performed as expected by the enclave.

Software Development Kit

The *Software Development Kit* (SDK) provides the developers with everything they need to develop an SGX-enabled application. It is composed of a tool to generate the interface functions between the application and the enclave, a tool to sign the enclave before using it, a tool to debug it, and a last tool to measure performance. It also contains templates and sample projects to develop an enclave using *Visual Studio* under Windows, or using *Makefiles* under Linux.

Shameless advertising: French readers will find more information on how to develop an application using Intel SGX in the upcoming issue of the MISC magazine.

Platform Software

The *Platform Software* (`PSW`) is the software stack that allows SGX-enabled applications to execute on the target platform. It is available for Windows and Linux operating systems, and is composed of 4 major parts:

- a driver that provides access to the hardware features;
- multiple support libraries for execution and attestation;
- the architectural enclaves necessary for the environment to run;
- a service to load and communicate with the enclaves.

Architectural Enclaves

To allow the secure environment to execute, several *Architectural Enclaves* (`AE`) are needed. They are provided and signed by Intel. They enforce launch policies, perform the provisioning and attestation processes, and even more.

Launch Enclave

The *Launch Enclave* (`LE`) is the enclave responsible for distributing `EINITTOKEN` structures to other enclaves wishing to execute on the platform. It checks the signature and identity of the enclave to see it if they are valid. To generate the tokens, it uses the *Launch Key* that it is the only enclave able to retrieve it.

Provisioning Enclave

The *Provisioning Enclave* (`PvE`) is the enclave responsible for retrieving the *Attestation Key* by communicating with the *Intel Provisioning Service* servers. In order to do so, it proves the authenticity of the platform using a certificate provided by the `PcE`.

Provisioning Certificate Enclave

The *Provisioning Certificate Enclave* (`PcE`) is the enclave responsible for signing the processor certificate destined to the `PvE`. In order to do so, it uses the *Provisioning Key* that it is the only enclave able to retrieve it. The `PvE` and `PcE` are currently implemented as a single enclave.

Quoting Enclave

The *Quoting Enclave* (`QE`) is the enclave responsible for providing trust in the identity of an enclave and the environment in which it executes during the remote attestation process. It decrypts the *Attestation Key* it receives from the `PvE`, and uses this key to transform a `REPORT` structure (locally verifiable) into a `QUOTE` structure (remotely verifiable).

Platform Service Enclaves

The *Platform Service Enclaves* (`PSE`) are architectural enclaves that offer other enclaves multiple services, like monotonous counters, trusted time, etc. Those enclaves make use of the *Management Engine* (`ME`), an isolated and supposedly secure co-processor that manages the platform.

Keys Directory

Each CPU with support for SGX contains two root keys which are stored inside e-fuses: the *Root Provisioning Key* (`RPK`) and the *Root Seal Key* (`RSK`). The `RPK` is known to Intel to enable the remote attestation process, while the `RSK` is only known to the platform. Despite the attacker model of SGX excluding physical attacks, efforts have been made to make the processor architecture difficult to temper with, or at least to make extracting the keys a very costly operation. With enough hardware, it is possible to read the e-fuse, but in a destructive way. That is why only an encrypted version of the keys is stored on the e-fuses. A *Physical Unclonable Function* (`PUF`) is used to store the symmetric key that is used to decipher the other keys during the processor execution.

Root Keys

Root Provisioning Key

The first key created by Intel during the manufacturing process is the *Root Provisioning Key* (RPK). This key is generated randomly on a dedicated *Hardware Security Module* (HSM) located inside a facility called the *Intel Key Generation Facility* (iKGF). Intel is responsible for maintaining a database containing all keys produced by the HSM. The RPKs are sent to multiple production facilities to be embedded inside the processors e-fuses.

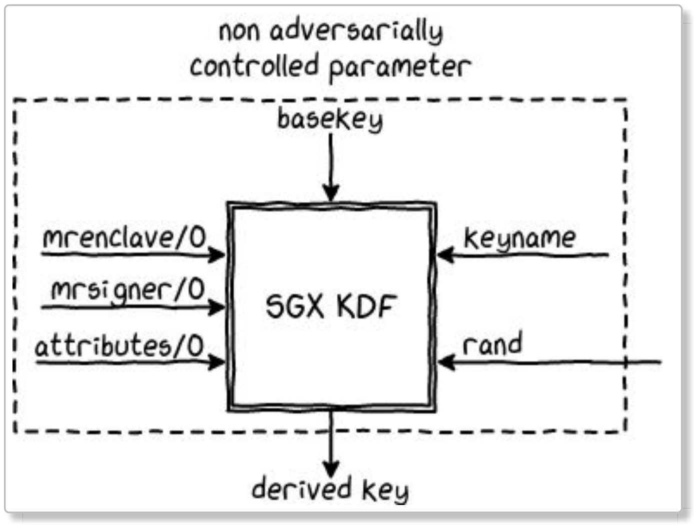
Root Sealing Key

The second key located inside the e-fuses is called the *Root Sealing Key* (RSK). Like the first key, it is guaranteed to differ statistically between each unit produced. Contrary to the RSK, Intel declares erasing every trace of theses keys from their production chain, in order for each platform to have a unique key only known by itself.

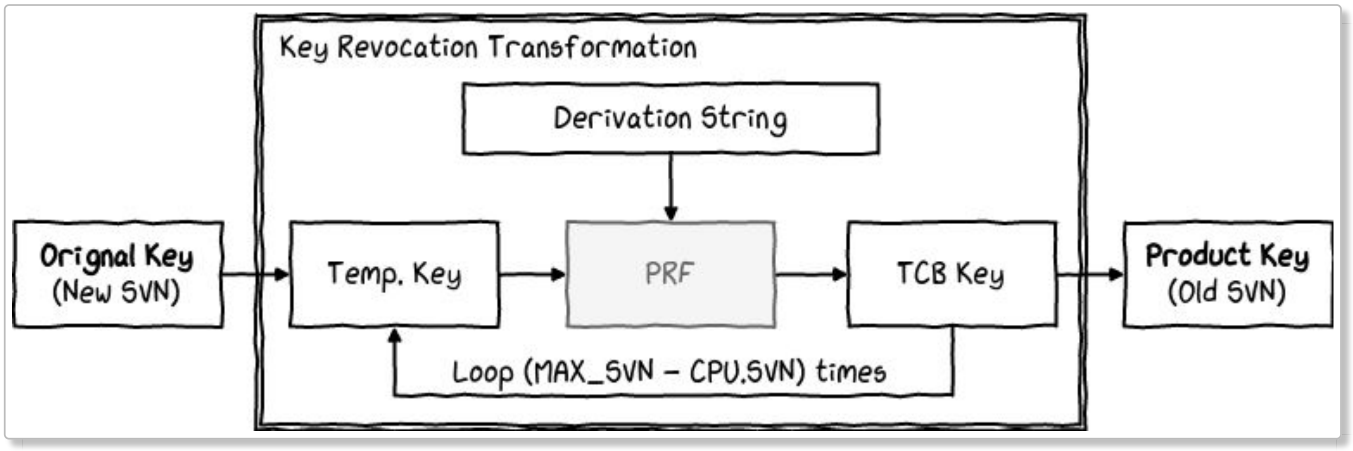
Key Derivation

By design, an enclave does not have access to the root keys. Nevertheless, it can access keys derived from the root keys. The derivation function allows an enclave author to specify a key derivation policy. These policies allow the use of trusted values like the `MRENCLAVE`, `MRSIGNER` and/or the attributes of the enclave. Enclaves cannot derive keys belonging to a `MRENCLAVE` or `MRSIGNER` of another enclave. Furthermore, when the key derivation policy does not make use of a field, it is automatically set to zero. As a result, even when non-specialized keys are available, specialized keys cannot be derived from them.

To add entropy coming from the user, a value called *Owner Epoch* is used as a parameter during the derivation. This value is configured at boot-time by the derivation of a password, and saved during each power cycle in a non-volatile memory. This value must stay the same for an enclave to be able to retrieve the same keys. On the contrary, this value must be changed when the platform owner changes because it prevents the new owner to access the personal information of the old owner until the original password is restored.



The SGX infrastructure supports TCB updates of its hardware and software components. Each component has a SVN which is incremented after each



Derived Keys

Provisioning Key

This key is derived from the RPK and used as a root of trust (tied to the TCB version) between the *Intel Provisioning Service* and the processor. As admitting a non-SGX processor into a group of legitimate SGX processors compromise the remote attestation for all processors, extreme precautions must be taken to disallow access to the *Provisioning Key*. Currently, the *Launch Enclave* gives access to this key only if the enclave is signed by Intel (the `MRSIGNER` of Intel is hard-coded in the *Launch Enclave* code).

Provisioning Seal Key

This key is derived from the RPK and RSK. During the enrollment of a processor in the group, the private key of each platform is encrypted with this key and sent to the *Intel Attestation Service*. It must be noted that the private key cannot only be encrypted using the RPK because that would destroy the anonymous enrollment protocol used. Similarly, the private key cannot be encrypted only using the RSK as it would allow non-privileged enclaves to access the private key of the platform. Unfortunately, known the uncertainty there is on the generation process of the RSK, one might assume that Intel knows the private key of each platform.

Launch Key

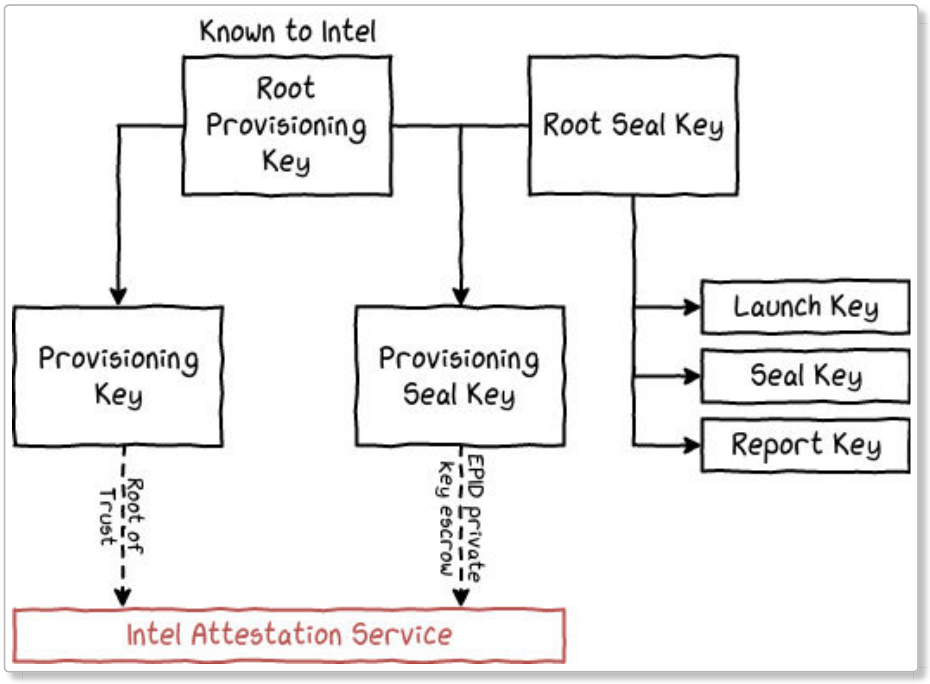
This key is derived from the RSK and is used by the *Launch Enclave* to generate an `EINITTOKEN`. Each enclave that is not signed by Intel must obtain this token otherwise the processor cannot instantiate it. Only a specific `MRSIGNER` - whose corresponding private keys are only known to Intel - can access the *Launch Key*. In SGXv2, the `MRSIGNER` of the *Launch Enclave* can be changed programmatically, but it is not known yet how Intel intends to apply access control to the *Provisioning Key*.

Seal Key

This key is derived from the RSK and used to encrypt data related to the current platform. It is important to not use a non-specialized *Seal Key* - either for encryption or authentication - because that would compromise the enclave's security.

Report Key

This key is derived from the RSK and is used for the local attestation process.

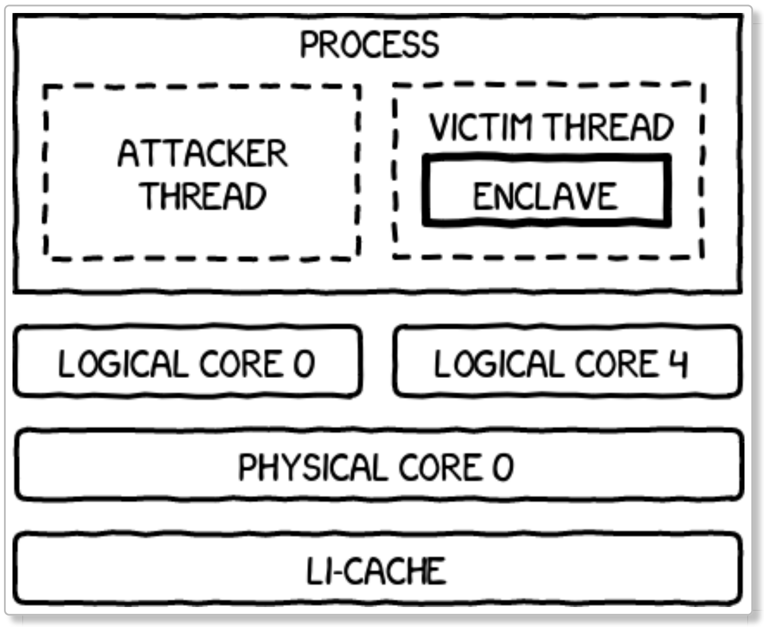


Side-channel Attacks

Intel SGX has been notorious over the years for its lack of resilience to various side-channel attacks. While Intel always warned that enclaves must be written in a way preventing side-channel attacks, a perfectly secure technology would not impose such a constraint on enclave developers. Below are summarized some attacks on Intel SGX to give the reader an overview of what is required to perform such attacks, and what information can be gained from executing them.

Cache Attacks on Intel SGX

To perform a cache timing attack on an Intel SGX enclave, the authors have pinned two kernels threads to two logical cores sharing the same physical core and L1-cache (see the figure below). The victim thread is running a version of an algorithm vulnerable to cache attacks inside an enclave. Probing a cache line is done using the `RDPMC` instruction, which requires a counter to be started in supervisor mode, which is totally within SGX security model.



Communication with the enclave is performed using shared memory instead of `ECALL/OCALL` as not to have to context switch from the running enclave. This is where the attack distances itself from a real-world scenario. `ECALL/OCALL` would cause the entire cache to be evicted, preventing the attack to work.

The attacker thread cannot be in a different process because process context switching requires to update the *Page Table* (PT), so the `CR3` register containing the base address of the PR would have to be changed, which triggers a TLB and L1-cache flush.

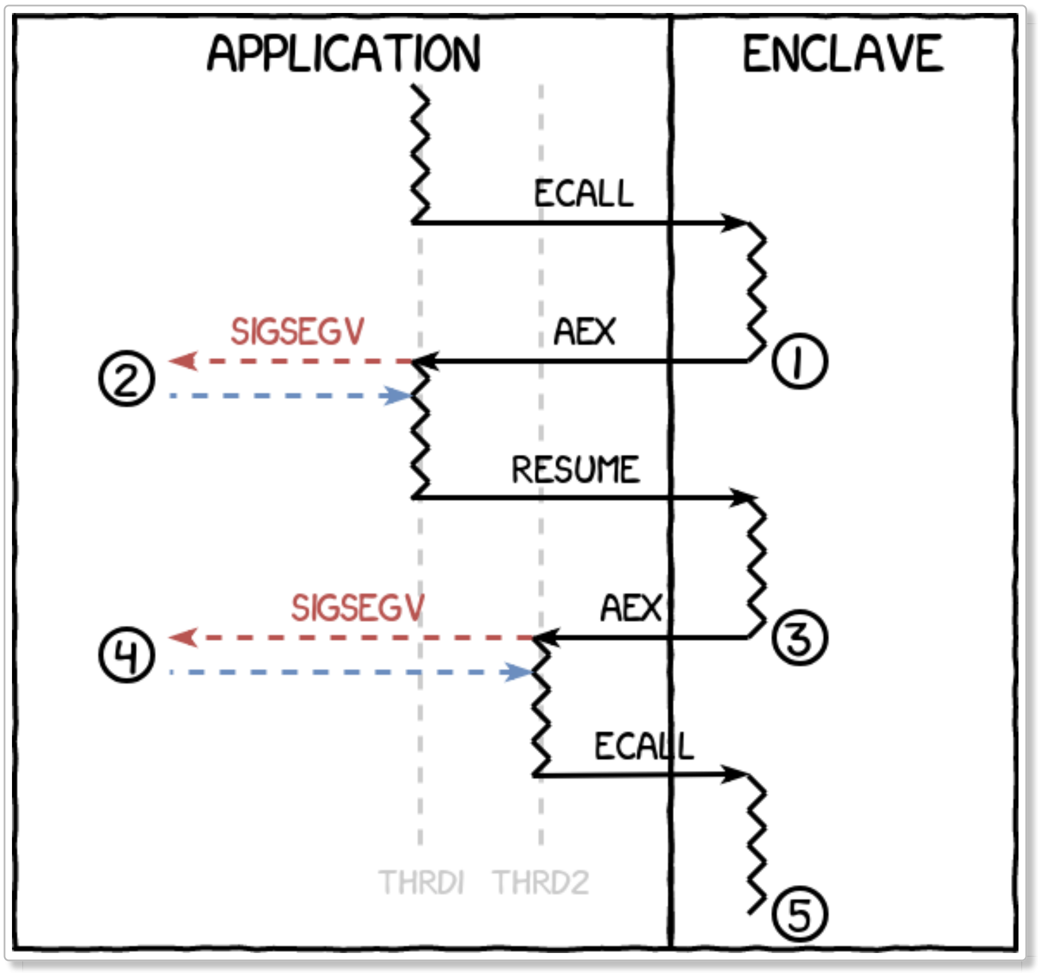
The AES implementation of the SGX SDK is not vulnerable to such side-channel attacks.

[1] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In Proceedings of the 10th European Workshop on Systems Security (EuroSec'17). ACM. 2017. http://www.sharcs-project.eu/m/documents/papers/a02-gotzfried.pdf

AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves

The idea behind *AsyncShock* is to facilitate the exploitation of existing synchronization bugs inside an enclave. In particular, it helps exploiting *Use After Free* (UAF) and *Time Of Check Time Of Use* (TOCTOU) issues. An attacker who is in control of the platform, which is within the threat model of Intel SGX, can interrupt enclave threads whenever he desires.

Interrupting a thread is done using the `mprotect(2)` system call to remove the read and execute permissions on a page. Because the traditional page walk is performed before checking if the access to an EPC page is allowed, the application can learn which memory pages the enclave is allowed to access (even though it has no way to see what the pages contain). When the thread exits the enclave, the execution resumes in the appropriate handler in the untrusted environment.



The application starts a first thread that is allowed to execute. The read and execute permissions are removed from the code page containing the `free(3)` function. When the thread calls `free(3)` (1), an access violation occurs, resulting in an AEX and a segmentation fault caught by the application (2). The permissions are restored for this page, but removed for the page containing the calling instruction, before the thread is allowed to continue. When the next marked page is hit (3), resulting in another AEX and segmentation fault (4), it signals that `free(3)` has returned. In the signal handler, the permissions are restored again. The first thread is stopped and a second thread is started and enters the enclave (5). The sample code for this example can be found below:

```
// Thread 1 enters the enclave
...
free(pointer);
// Thread 1 is interrupted, exits the enclave
pointer = NULL;
...

// Thread 2 enters the enclave
...
if (pointer != NULL) {
    // Thread 2 uses a pointer to freed memory
}
...
```

Combined with a function pointer inside a structure and a memory allocator that reuses freed memory for the new allocations, these kind of bugs have the potential to allow for *Remote Code Execution* (RCE). TOCTOU bugs might allow incorrect parameters to be used in the enclave, which might also have huge security implications. Consider the simple example below:

```
...
/*  1 */ static int g_index = 0;
/*  2 */ static int g_value = 0;
/*  3 */ static int g_array[256];
/*  4 */
/*  5 */ void ocall_set_index(int index) {
/*  6 */     g_index = g_index;
/*  7 */ }
/*  8 */
/*  9 */ void ocall_set_value(int value) {
/* 10 */     if (g_index < sizeof(g_array)) {
/* 11 */         g_array[g_index] = value;
/* 12 */     }
/* 13 */ }
...
```

It may appear that `g_array` cannot be accessed with an invalid `g_index`. A first thread can execute lines 9 and 10 of the `ocall_set_value` function and then get interrupted. A second thread can then execute lines 5 to 7 of `ocall_set_index` to change the value of `g_index` after it has been checked by the first thread. The first thread can then be resumed and the access performed at line 11 will be done with the value of `g_index` set by the second thread. This results in a *Out Of Bounds* (OOB) access.

These attacks require complete control over the platform, knowing what code is running inside the enclave, and having found synchronization bugs in it. The best protection against this attack is to disable multithreading inside the enclave, but it will obviously hinder the performance of the program. Another solution might be to encrypt the code of the enclave and use the remote attestation process to provide the enclave with the key needed to decrypt its code.

[2] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In: ESORICS 2016. Lecture Notes in Computer Science, vol 9878. Springer. 2016. https://www.ibr.cs.tu-bs.de/users/weichbr/papers/esorics2016.pdf

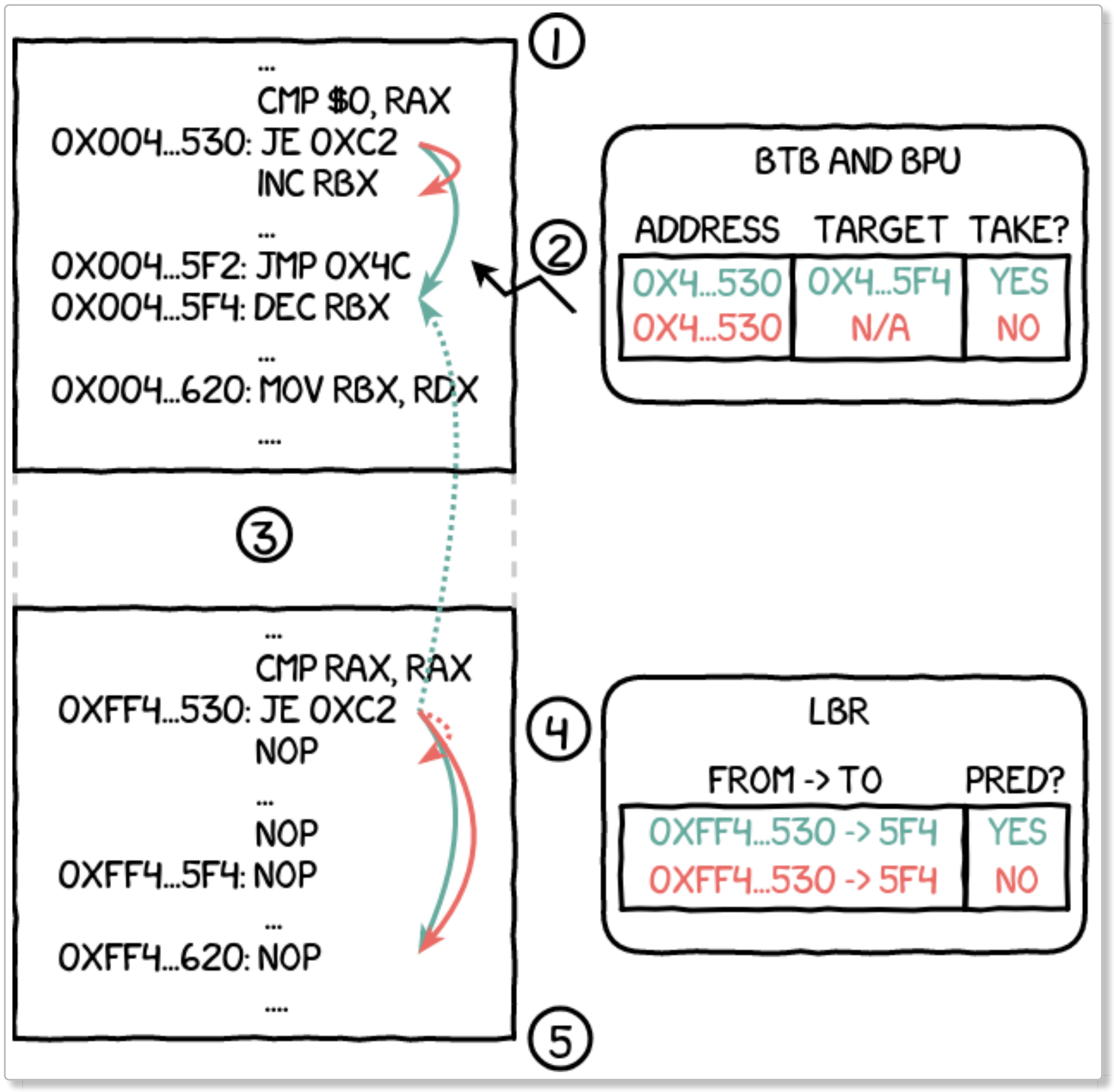
Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing

Inside the CPU, the *Branch Prediction Unit* (BPU) uses the *Branch Target Buffer* (BTB) to log information useful for branching predictions. While this information is only used internally by the processor, Intel SGX leaves its branch history uncleared during enclave mode switches. This allows the branches taken (or not) within an enclave to influence the prediction of the branches outside the enclave. A technique called *branch shadowing* was developed to infer the control flow of a running enclave.

The idea is to replicate the control flow of an enclave program inside the untrusted environment, carefully choosing the address at which this new code is mapped in order to introduce collisions inside the BTB. By executing the branch within the enclave code first, then within the shadowed code, the prediction of the second branch is affected by the result of the first. To know what was predicted by the CPU, the *Last Branch Record* (LBR) can be used only in the untrusted environment, as it is disabled for enclaves.

In order for this attack to work, the enclave execution must be interrupted as frequently as possible to perform the necessary measurements to infer the control flow of the enclave. The APIC timer can be used to interrupt the execution every ~50 cycles and, if more precision is needed, disabling the CPU cache allows to interrupt up to every ~5 cycles.

Below is an explanation of the detection of conditional branches occurring within an enclave (green represents the case where the branch is taken, red where it is not):



1. The conditional branch instruction of the enclave is executed. If taken, the corresponding information is stored within the BTB. Because this is occurring within the enclave, the LBR does not report this information.
2. Enclave execution is interrupted by the APIC timer and the OS takes control.
3. The OS enables the LBR and executes the shadowed code.
4. If the branch in the enclave was taken, the BPU correctly predicts that the branch will be taken, even though the target address is invalid because it is within the enclave. If it was not taken, the BPU incorrectly predicts that the branch will not be taken.
5. By disabling and retrieving the LBR content, the OS can learn whether the enclave branch was taken or not by checking if the shadowed conditional branch was correctly predicted.

Similar techniques are presented by the paper's authors to detect unconditional branches and indirect branches (the target address cannot be recovered by the attack). This attack requires complete control over the platform, and knowledge of the code being executed inside the enclave. It also introduces a significant slowdown that an enclave might be able to detect (but it is not as simple as executing RDTSC because it is not allowed inside enclaves).

[3] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17), USENIX. 2017. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-lee-sangho.pdf

Concerns

The first concern expressed by SGX users is that they must put trust in Intel. Notably, Intel says that it does not retrain the RPK embedded in each CPU die at their manufacturing facilities. But if it turns out that they did, in any way possible, it would invalidate the security of every platform. Furthermore, enclaves signed by Intel are granted special privileges, like the *Launch Enclave* (LE) which is used to whitelist which enclaves are allowed to execute. Developers need to register into Intel programs to be able to sign release version of enclaves. There exists an open-source initiative to develop an alternative LE, that should be allowed to replace the current one starting with SGXv2.

The second concern is that it would be possible for a malware to execute its malicious code inside an enclave, basically protected from everything and everyone. However, it is important to keep in mind that the code within an enclave has no I/O. It relies solely on its accompanying application for access to the network, file system, et cetera. Therefore, technically, analyzing an application can tell you a lot about what an enclave can do to a system, mitigating the fear of a "protected malicious code running inside an enclave". On the opposite side, the lack of trusted I/O is a problem for securing user information and some work has already been done on this subject, with proprietary solutions like the Protected Audio Video Path (PAVP) and academic like SGXIO.

The third concern was about the impact of Meltdown and Spectre on SGX enclaves. While they are not vulnerable to the former, the authors of the SgxPectre paper have demonstrated that variants of Spectre allowed to read enclave memory and register values. This enabled the recovery of the *Seal Key* of the platform, and consequently of the *Attestation Key*, effectively bypassing the whole security offered by SGX. Intel released a micro-code update to prevent these attacks and, thanks to the *Security Version Number* (SVN), it is also able to ensure that those patches have been applied in order to pass the remote attestation process. A new attack called SpectreRSB has recently been released and it is able to by-pass the patches by targeting the *Return Stack Buffer* (RSB) instead of the BTB. Intel will have to release another micro-code update to fix this new issue.

Conclusion

Intel SGX is a promising technology which is still in its infancy. It fulfills the needs of the *Trusted Computing* industry, allowing an enclave to be protected from the other pieces of software executing on the platform, and in limited ways, from hard tempering. It does so while giving the platform owner some degree of control over the execution to allow for resources management. The attestation process allows secrets to be securely transmitted to the enclave. Developing an SGX-enabled application is made really easy thanks to the provided SDK.

However, there remain some issues with the current iteration of Intel SGX. Being prone to side-channel attacks sadly limits the security offered by the platform, forcing developers to proactively ensure that their programs cannot be attacked. In a perfect world, developers would not need to worry about such considerations and SGX should ensure that these attacks are impossible. These attacks can all be prevented by encrypting the enclave's code and proving the key via remote attestation, but this is an inconvenience. Finally, users have to put an enormous faith into Intel, and while they have demonstrated a perfect incident response so far, uncertainties still remain.

Acknowledgements

- Sébastien Kaczmarek and Romain Thomas for supervising me;
- Joffrey Guilbon and Paul Hernault for their thoughtful advice;
- My other Quarkslab colleagues for proofreading this blog post.

Comments

1 Comment Quarkslab Disqus' Privacy Policy 1 Login

Recommend 2 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Sindhoor Tilak • 2 years ago

Brilliant blog post. Hoping for more articles on side channel attacks and TEE's

Thanks for keeping it open!

Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

Powered by Pelican, Theme is from Bootstrap from Twitter