Supply chain malware targets SGX: Take care of what you sign

Andrei Mogage
Alexandru Ioan Cuza University of Iaşi
Iaşi, Romania
mogage.andrei.catalin@info.uaic.ro

Rafael Pires
University of Neuchâtel
Neuchâtel, Switzerland
rafael.pires@unine.ch

Vlad Crăciun
Alexandru Ioan Cuza University of Iaşi
Iaşi, Romania
vcraciun@info.uaic.ro

Emanuel Onica
Alexandru Ioan Cuza University of Iaşi
Iaşi, Romania
eonica@info.uaic.ro

Pascal Felber
University of Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

***Abstract*—Malware attacks represent a significant part of today's security threats. Software guard extensions (SGX) are a set of hardware instructions introduced by Intel in their recent lines of processors that are intended to provide a secure execution environment for user-developed applications. To our knowledge, there was no serious attempt yet to overcome the SGX protection by leveraging the software supply chain infrastructure, such as weaknesses in the development, build or signing servers. While SGX protection does not specifically take into consideration such threats, we show in the current paper that a simple malware attack exploiting a separation between the build and signing processes can have a serious damaging impact, practically nullifying the SGX integrity protection measures. Finally, we also suggest some possible mitigations against the attack.**

***Index Terms*—security, dependable software, supply chain, malware, SGX.**

I. Introduction

A software supply chain attack can be informally defined as the act of compromising legit software packages during their development or distribution phases. The number of such attacks showed a tremendous increase over the last few years, including high impact ones. A recent NIST forum presentation [1] reported seven significant events in 2017 compared to only four during the previous three years. One of the most common attack vectors is injecting malicious malware code [1], [2] into legitimate software packages during or between development and distribution phases, such as upon building or signing. The most prominent example is an infected installation package of the well known CCleaner [3] application that included a malware deployed in the vendor's build server [4]. The altered binary file was downloaded by 2.27 million customers, with potentially serious effects ranging from keystrokes recording to stealing secret credentials from users.

There are also other recent documented examples of fairly similar supply chain attacks, such as an embedded malware in software packages released by NetSarang [5], a company that develops secure connectivity solutions, or corrupted packages injected with malicious code used for updates on M.E.Doc [6], a highly popular accounting application suite in Ukraine. The focus of our paper lies on the severe implications in a supply chain attack scenario against one of the most recent approaches of preserving confidentiality and integrity of applications: Intel SGX.

SGX [7] is a set of instruction extensions introduced by Intel in their line of commodity processors since the Skylake generation in 2015. SGX offers developers the benefit of a trusted execution environment (TEE) supported in hardware for critical applications or parts of applications requiring enhanced security levels. The TEE can be used as an isolated space for executing code in *enclave* containers where confidentiality and integrity are assured. The integrity of a software application that is supposed to run in an isolated enclave is determined based on a *measurement* that uniquely identifies the software code inside the container [7]. This measurement is computed as a hash when the enclave is initiated. Based on this value, an *attestation* procedure can be performed whenever a third party wants to check if the correct code is actually running in a SGX-capable machine. This check implies that each time the enclave is loaded for execution, the measurement is re-calculated and compared for integrity against the initial value, obtained at signing time. If any steps of the enclave building process are altered, resulting in alterations of the loaded code and data, this integrity check will fail.

Due to the execution in a guaranteed isolated secure environment, the SGX enclave integrity provisioning is a very attractive countermeasure against previously described supply chain attacks. If a critical software application is loaded within a secure enclave, its alteration through malware embedding could be easily detected as part of the attestation process. Unfortunately, as we prove in our paper, an attacker can currently circumvent the SGX integrity protection using a particular attack methodology that implies injecting the malware between the time of building the target software binary and its signing phase that prepares it for the attestation.

Although, in some light, the attack we show can be regarded as generic, since it is applicable to any software package that

2575-8462/19/$31.00 ©2019 IEEE
DOI 10.1109/SRDS47363.2019.00016

is vulnerable to tampering before applying a secure signature, the case we present has particular severe implications on the attempt to protect integrity using SGX. The attack targets the SGX signing process and renders useless its enclave measurement in the way this is initially computed, as well as all its subsequent verifications. As a result, this means that developers and users cannot blindly trust SGX in itself as a way to protect their code and data, unless the signing process that includes the enclave measurement has been conducted in secure environment. To counter the attack we propose securing this measurement at the enclave compilation phase. We also advocate towards some variants for a secure topology that would bring real benefit in using SGX for protecting software integrity.

In Section II we provide details over the measurement mechanism and the way this is currently used in verifying the integrity of an enclave. Section III describes the actual attack scenario, pointing out current vulnerabilities. Section IV provides some practical details regarding the effective attack implementation. Section V discusses possible ways of mitigating the attack. In Section VI we provide a brief overview of related work and we conclude in Section VII.

II. SGX Background

SGX has been available as a TEE in Intel processors since the Skylake family. It was intended to allow applications to safely handle sensitive data when running within secure *enclaves* against an attacker who has full control of the operating system (OS). The security boundary is the central processing unit (CPU) die, where data is available in plaintext form. Outside it, enclaves' data are always encrypted and their respective digests are kept for integrity and freshness checks.

One SGX application is formed by the combination of two logical components: trusted and untrusted. Untrusted code runs in user mode, it is responsible for asking the operating system to allocate enclave memory and is able to perform enclave calls (ecalls) through a special instruction, but it does not have access to enclave memory pages. Trusted code, on the other hand, is able to access both its own pages and the ones that belong to the same process running in untrusted mode. Since the OS is not part of the trusted computing base (TCB), the enclave is not able to directly perform *system calls*, and it can only execute these through outside calls (ocalls). Since trusted code deals with sensitive data, it is signed and can be *attested* by local or remote parties before being provisioned with secrets.

The development of applications targeted to run within SGX enclaves, besides the usual iterations on coding and compiling, also includes a mandatory signing step before the executables are able to be deployed and used in production. This serves two essential purposes: *(i)* the code is uniquely associated to independent software vendors (ISVs), making them accountable for any consequence originated from their product; and *(ii)* whoever communicates with the application can have guarantees that the enclaved endpoint has loaded and is actually running the expected code within a genuine SGX platform.

Intel offers to ISVs two signing methods [8]: *(i)* single-step method, for development or pre-release modes, which uses a test private key locally stored in the building system; and *(ii)* two-step method, for release enclaves made by ISVs who have obtained a *production license* from Intel [9]. With the two-step method, the ISVs first generate the *signing material*, which is later signed in a different facility that has access to the signing key. Then, the signature comes back to the building platform and is appended to the enclave's metadata. Figure 1 illustrates the two-step method.

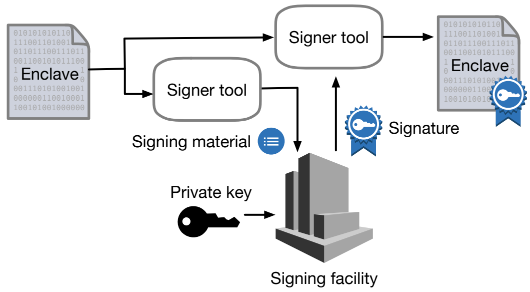

Fig. 1. Two-step signing method

The signing material includes information about the vendor, the date, some attributes, a version number and, especially important to our attack, the enclave *measurement* hash. This hash corresponds to a digest made upon the enclave's initial state, including data, code and metadata [7]. When the enclave is loaded, a hardware implementation of the same procedure performs a *measurement* on the actual content of the running enclave, which has to precisely match the one that was computed during the signing step. Tamper attempts would be detectable by this protection scheme. Our attack, however, acts before the signing material is generated and therefore passes undetectable by the *measurement* comparison.

Later on, when interlocutors want to communicate with a running enclave, they should first attest it before sharing sensitive data with it. Enclave attestation can be performed locally or remotely, the latter being dependent on the first. The attestation procedure starts locally through a previously established communication channel, when the attestor—which is a platform enclave in case of a remote attestation—sends its identity (*measurement*) to the enclave being attested. This, in turn, calls a special *report* instruction that cryptographically binds the enclave measurement with other security-related information. This report's signature can only be checked locally by another enclave, as it is generated with a hidden key embedded in the platform. In case of remote attestation, another report called *quote* must be generated. This quote, in turn, may be checked by the remote party with the aid of Intel attestation service (IAS). Since all this happens after our attack has been performed, the measurement will correspond to that of the tampered enclave, and therefore it will pass all checks.

III. Attack Scenario

We consider a context where an attacker can gain access to a machine where the *signing material* is generated for an SGX enclave, as described in Section II. Frequently, this is executed on the build server where the enclave is compiled. We assume that the attacker is able to deploy a malware on such machine. As referenced in the introduction this is a plausible scenario, multiple similar cases being recorded in the recent period.

The malware will intercept the process that receives the enclave binary as input and generates the *signing material*. We further refer to this process as the *signer* process. The malware will suspend the *signer* process and patch the enclave with malicious code. Finally, the *signer* process is resumed. The generated *signing material* will include the enclave *measurement* computed over the tampered enclave. The effective enclave signature will be applied on it as nothing abnormal would have happened. Since the enclave integrity assurance is based on the comparison between the signed measurement and the actual loaded content, any further integrity checks on the maliciously patched enclave will succeed.

Figure 2 depicts the usual chain for manufacturing an enclave and pinpoints where our malware attacks: after the binary is produced by a compiler and before it is signed. Note that the figure is representative for the single-step signing method, where the signing material generation and the effective signing are part of the same process, but the flow is similar for the two-step method, with the difference that a separate process performs the effective signing, as shown in Figure 1.

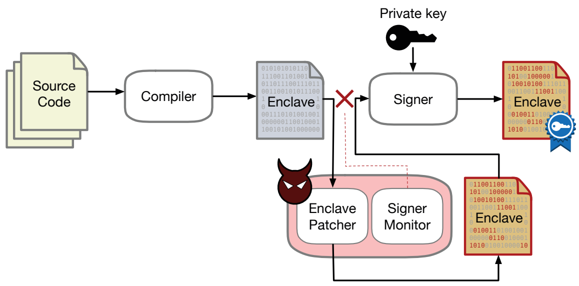

Fig. 2. Attack flow architecture

The malware includes two components used for hijacking and infecting the enclave manufacturing chain: the signer monitor and the enclave patcher. The execution flow of the two components is illustrated in Figure 3.

The *signer monitor* has the role of scanning the processes that are currently running in the machine until it is able to identify and suspend the signer process. To achieve that, it uses heuristics such as the process name, input parameters, memory occupancy, hash on certain memory chunks or digital signature.

The *enclave patcher* is composed by malicious code and instructions on where to inject this code. This depends on the attacker's knowledge about the enclave, ranging from very specific changes on its behavior, like altering some remote server endpoint address, to more generic approaches, like exfiltrating as much information as it can. We exemplify in the following section a use case where such a malicious patch is hooked to the functions listed in the enclave's ecall table, leaks sensitive data out of the enclave and changes data within the secure enclave space.

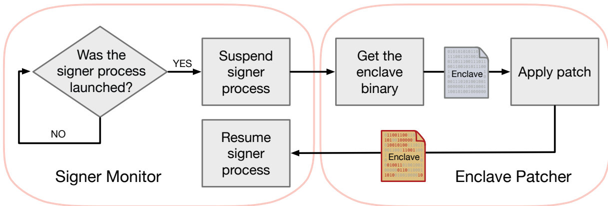

Fig. 3. Malware operation

IV. Use Case and Implementation Details

We describe a practical use case about how an attacker could profit from the window of opportunity detailed in Section III for getting access to sensitive data and changing it. Concisely, we inject code to learn about internal data structures and monitor their content before being able to modify a piece of sensitive data. Besides, we describe a data exfiltration patch.

SGX enclaves are accessed through an instruction called `EENTER` that transfers the execution to a single entry point in the protected area. The specific ecall routine address is then fetched from a table to which we refer as *ecall table*. The enclave patcher finds this table by a series of steps illustrated in Figure 4. First, the enclave dynamic link library (DLL) is disassembled with the aid of the BeaEngine library [10]. It is responsible for parsing and interpreting the portable executable (PE) format in which the DLL is organized. The enclave's export data section contains a symbol called `enclave_entry`, which is associated to its entry point address (❶). By following this address, we find a piece of code that occasionally executes a `call` instruction to a given address (❷). When followed (❸), this address leads to a chain of other calls to pieces of code that are similar across different enclaves. Eventually, the ecall table is consulted (❹). In all of our enclave samples, the ecall table was located somewhere in the read-only initialized data section (`.rdata`) of the DLL. Once we find the table, a similar procedure happens to find the ecall function implemented by the enclave developer (❺ and ❻), since the SGX software development kit (SDK) adds some wrappers in order to perform security checks before calling the actual enclave code. All these heuristics were obtained through the analysis of several different enclaves generated by the SGX SDK version 2.0.101.44281, until a set of patterns allowed us to find the ecall table with certainty. Although this construction may change across different SDK versions, the attacker could apply a distinct set of heuristics depending on the version, which is explicitly marked in the DLL's metadata.

Once the enclave patcher finds the target ecall, it injects a jump instruction in its beginning to a specially crafted

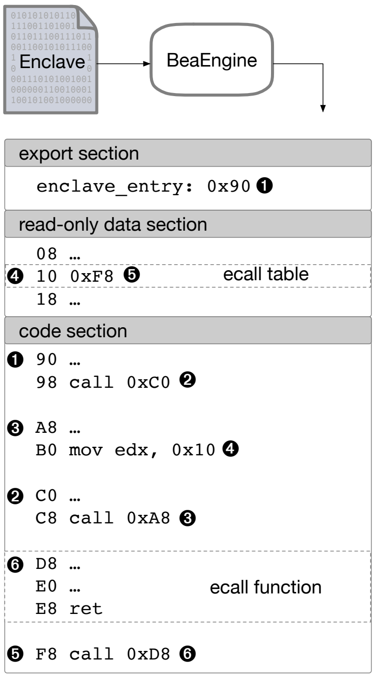

Fig. 4. Steps to find ecall table

piece of code. We refer to it as the *patch*, whose address is marked with the label `HOOK`. Besides the jump instruction to the patch, we add a label (`BACK`) where the execution continues after executing the hooked code. Figure 5 shows on the left the initial state of the ecall table and the functions it points to. In the bottom, we depict a set of disjoint chunks of free memory within executable pages. They are found based on their content, which can be contiguous areas containing only zeros or ones, as these could not possibly refer to any instruction codes. Such areas are used for placing the patch code. Since there might not be a single chunk big enough for holding the malicious code, the patcher may break it into several pieces linked by jumps and labels. On the right side of the figure, we illustrate the enclave after applying the patch. The hooked code is split in two and connected by the label `H1`.

As for the patch code, we first describe the exfiltration example, depicted as the *data leak patch* in Figure 6. The malicious code first tries to identify the arguments of the ecall function by looking at the stack. It checks, among the parameters, if there is an output buffer by evaluating if the pointer refers to an untrusted piece of memory. In the figure, this pointer is referred to as `Ptr0` and it will be used by the patch to leak sensitive data. The output buffer `Ptr0` is shown between two other arguments passed by value: `Val0` and `Val1`, whose contents the attacker is interested in leaking along with other local variables that happen to be in the stack.

In our prototype, the hook is injected in the beginning of the ecall. One might argue that these values are not interesting for exfiltration at this point, since they are the exact same as what was given as parameter from the untrusted code and therefore the attacker already knew them. Yet, placing the hook in the end of the function would not be a good idea, since at this point the ecall would have already written the output buffer and it would not be possible to use it as the data leakage vector anymore. So, the ideal placement is after some computation on local variables has been done, but before the output buffer is written. For simplicity, we chose to place the hook in the beginning. We point out, however, that a more useful stack data leakage attack would do it differently.

Once the control is diverted to the patch, it copies the stack data to the output buffer. To confirm that the data leakage has happened and facilitate the location of it, the patch also prepends a marker in the output buffer, referred to as `MALW` in Figure 6. To prevent that the output buffer be overwritten when the ecall function is resumed, the patch also uses a spin lock on a boolean variable in the output buffer before it gives back the control to the ecall. Once the untrusted part reads the leaked content, it changes the value of this variable and lets the enclave execution go on. The leaked data is shown in the first "hexdump chunk" of Figure 7, preceded by the marker. Although this is an illustrative example, the same techniques could be used for leaking session keys, server credentials or actual payloads decrypted inside the enclave.

Our second experiment, instead of just leaking information, also changes it. We used the remote attestation end-to-end example [11] and the signing tool [12], both provided by Intel. It basically performs all the necessary steps for remotely attesting a server and establishes a session key. Our tampered binaries passed undetected by all attestations, as expected. We slightly modified the server by adding the transmission of supposedly sensitive information encrypted with the session key, the string `"John;892157932877159;$100"` symbolizing, for instance, the destination of some financial transaction.

The enclave patcher, in this case, includes the trampoline to the patch in the end of the decryption function, so that we can modify the information that arrived from the remote server right after it was deciphered with the session key. The two hexdump chunks of memory on the bottom of Figure 7 show the tampering, by replacing "John" for "Lary". Note that we write plaintext decrypted content in the output buffer provided as a parameter of the ecall from the former example. This happens to be untrusted memory area, accessible by the attacker. In a real world application, any sensitive content must only leave the protected memory in encrypted form. For the sake of this experiment, however, we used untrusted memory for being able to monitor the tampering when it happened.

In general, our approach is similar to any infection of executable files, where malicious code is hooked on the execution flow and the injection is done in the free space within the executable section. Nevertheless, we also consider some specificities of SGX in our design, like analyzing the ecall table, testing whether pointers belong to trusted or untrusted memory areas and synchronizing with untrusted code through

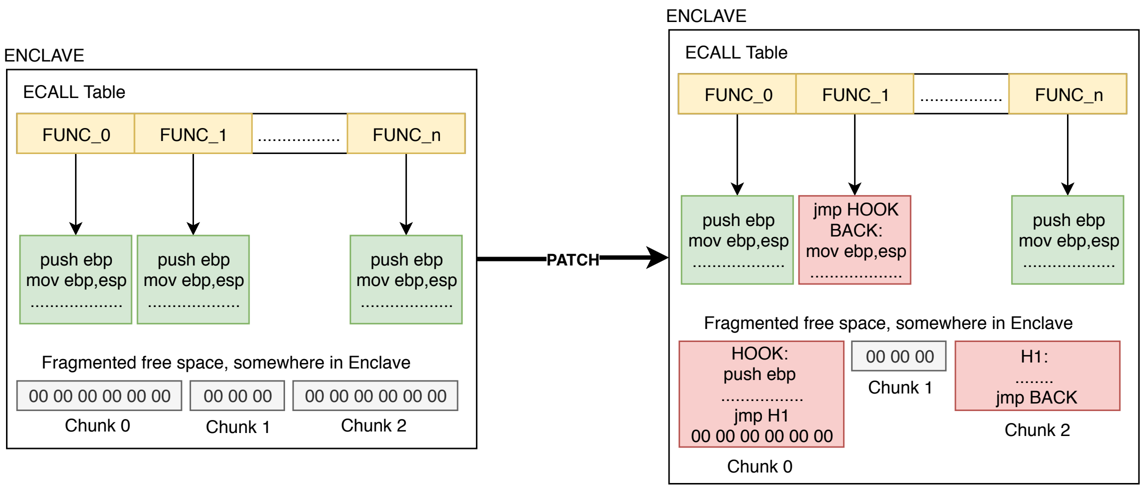

Fig. 5. Enclave patcher

spin locks.

With regards to performance, we measured the time it takes to sign the enclave with and without activating our malware. Normal signing took on average 46.6 ms and 47.1 ms when applying the malware. Both experiments were performed 100 times, with negligible standard deviation. All experiments were conducted on a machine with an Intel Core i7-7700 CPU at 3.60 GHz, with 16 GB of RAM and using Windows Server 2019.

V. Mitigations and Discussion

Our attack happens entirely in user-mode and needs to strike where the *signing material* is generated (see Section II). The platform where the enclave binary was compiled or where it will be signed are irrelevant to the attack's accomplishment. This is why our approach would succeed even with the supposedly more secure two-step signing method [8], where the private signing key is distinctively protected in a different platform. Safely binding the binary produced by a legit compiler with the signing material and the signing process is precisely where mitigation actions should occur.

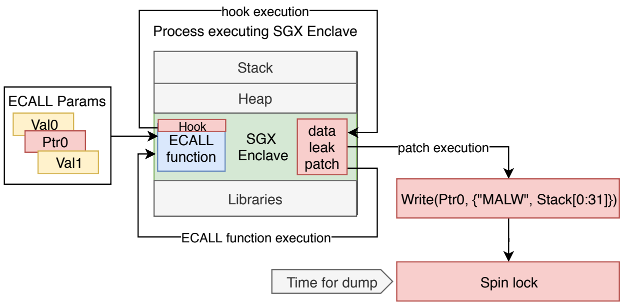

Fig. 6. Exfiltrating enclave data

Indeed, the single-step signing method would always expose the signing key to an attacker who has succeeded to inject a malware in the signing platform. Instead of trying to tamper with the enclave to be signed, attackers could just exfiltrate the key and impersonate the ISV by launching any number of rogue enclaves with genuine signature. Tampering with the enclave, however, stealthily puts the patch inside the vendor's production code, and potentially gives access to sensitive data more easily.

With regards to the malware's privileges, we can consider two possibilities: *(i)* it runs in user mode and any attempt to manipulate other processes is blocked by memory protection mechanisms; *(ii)* it can escalate to administrator super user, load kernel modules and manipulate code and data of any process.

In the first hypothesis, as a mitigation action, one could put compilation and signing in the kernel. Input of sensitive data, such as code and signing key, and output of signed material could happen through any input/output device (flash drive, network, etc.) and never reach user mode.

To mitigate the attack of one who has full control of the OS, we can envision two possible solutions: *(i)* centralized, using SGX protection; or *(ii)* distributed by relying on several nodes and assuming that a majority is not compromised. The centralized approach would put compilation and signing inside dedicated SGX enclaves. This special enclave would have been carefully crafted and accordingly signed by a trusted party. Data provisioning (code and key) would be performed through the typical SGX attestation and secret provisioning (see Section V-A). SGX memory constraints might seem to be a potential problem for the compilation requirements. This can be addressed with appropriate software caching management [13] or ordinary memory swapping. SGX version 2 supports dynamic memory loading [14]. In any case, compilation time is not the most concerning aspect for building a secure application. As for the distributed approach, one could

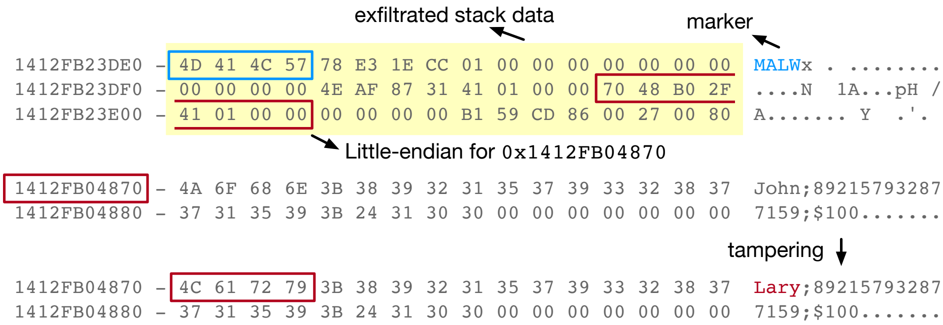

Fig. 7. Obtained data

resort to comparison and consensus [15], [16] on the enclave's hash, after each participant node had compiled its own copy of the source code. This approach would require that the attacker compromises a certain number of build servers, which is hard to accomplish.

Exposing the attack, however, is easier. The attack can always be detected if one compares the original untampered enclave with the final tampered and signed one. When the attacker leaves the original input binary untouched, such comparison can be performed after the signature stage. For the two-step signing method, on the other hand, the attacker must replace the input with the tampered version, since the signature comes later and it must match with the enclave binary, or the measurement comparison would fail when launching it. As a consequence, it is also possible to detect the attack at the signing material generation step. It is therefore of utmost importance to keep the original untampered file for successful detection of the attack. Comparing the input file before and after launching the signing tool is, however, arguably unusual, and only likely to be performed by the awareness of attacks such as ours. In conclusion, facility of exposure does not nullify the necessity to enhance supply chain security.

A. SGX compilation and signing

We further investigate the centralized mitigation approach described above by designing, implementing and measuring the performance of a combined compiler and signer within SGX enclaves. Such method would guarantee that the enclave signature corresponds to that of the binary generated by an accredited compiler even if the signing machine is compromised by a super user who has full control of the operating system. We assume, however, that the source code and private key were safely provided through encrypted channels after an attestation (see Section II) performed by the ISV, as illustrated in Figure 8. Once the enclave is attested and the secure communication channel is established (❶), the ISV provides the source code it intends to compile and sign (❷). The compiler embedded in the enclave then generates the binary and provides it to the signer (❸), which also resides in the secure environment. This, in turn, computes the signature of

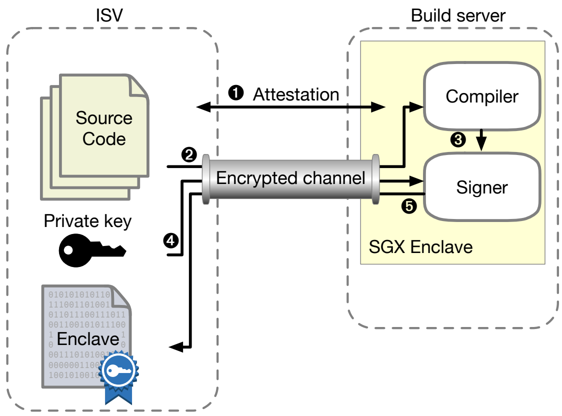

Fig. 8. SGX compilation and signing

the generated binary using the private key provisioned by the ISV (❹), to whom it finally sends the final signed enclave (❺).

We chose the tiny C compiler (TCC) [17] for turning source code into an executable image. TCC provides support for cross compiling Windows MZPE files (`.exe` applications and `.dll` dynamic libraries) and Linux extensible linking format (ELF) files (`a.out` executables and `.so` shared objects) for x86, x64 and ARM architectures. As in our previous experiments, we used Windows x64 as build target. Figure 9 portrays the data flow along with the adaptations we carried out. In order to support the compilation of legacy code, we had to ship with the enclave some common dependencies, such as standard libraries, headers and the common runtime (CRT). In spite of this, the final enclave size, including compiler and signer, accounted for only 2.14 MiB out of the 93.5 MiB of usable enclave page cache (EPC) memory [18].

The signer, depicted in Figure 10, receives the compiler output and the signing private key. We implemented the signature using crypto libraries available in the SGX SDK (*sgx_tcrypto* and *sgx_ippcp*). Particularly, secure hash algorithm 256-bit (SHA-256), and RSA 3072 bits (public exponent equal to 3), as specified by Intel [19]. The signature (encrypted hash) is then appended to the enclave binary along with the

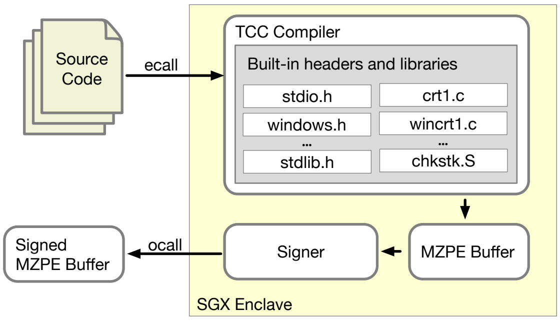

Fig. 9. TCC Compiler within an enclave

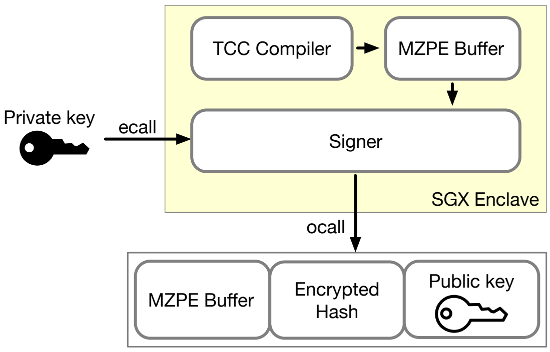

Fig. 10. Signer

corresponding public key. This bundle is then sent back to the ISV.

The system setup for measuring the compilation and signature durations is a machine equipped with an Intel processor i7-8650U at 1.90 GHz, with 16 GB of RAM and using Windows 10 Professional x64 build 1803. We tested our mitigation approach on 5 samples of C source code of different sizes. Table I provides a brief description of these samples. We compiled and signed the corresponding binaries both within a secured SGX enclave using the setup described above, as well as simulating the normal native enclave building steps, outside the secured enclave, in order to observe any potential downgrade in performance. Results are shown in Figure 11. Each experiment was repeated 11 times and averaged. Error bars correspond to the 95% confidence interval.

We can notice that even if compilation times take longer inside the enclave for most samples, the total time is either equivalent or smaller than natively doing the same. The main reason resides in a significant difference between our inside enclave implementation and the outside one. The outside version writes the binary on disk and reads it back for the signing step. Our SGX version, on the other hand, uses a memory buffer. Besides, the signing tool provided by Intel performs additional checks on the input file. Since in our experiments the compilation output never leaves the enclave, these checks are obviated.

In large projects, we can assume the whole EPC memory

TABLE I
SOURCE CODES USED IN THE BENCHMARK

Program	LoC	Description
hello	5	Simply prints a message
fibonacci	25	Computes the n^{th} fibonacci number
solitaire	418	Console version of *peg solitaire* puzzle
lisp	1,439	Lisp interpreter
malware	27,676	Malware described in Section IV

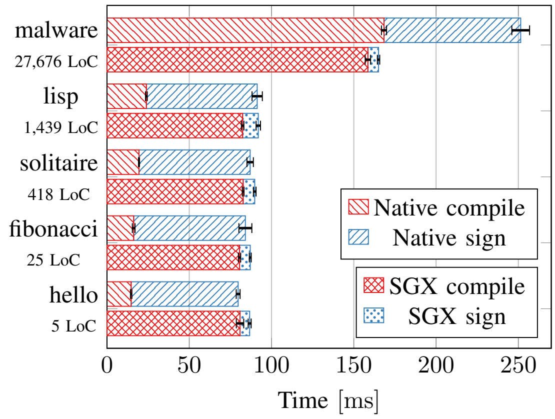

Fig. 11. Compilation and signing duration when varying source code sizes

could be exhausted, in which case we expect that the performance would degrade dramatically due to memory paging [20] or the necessity of writing temporary files on disk (in such case, the SGX sealing mechanisms should be used). Nevertheless, the trade-off between compilation time and security would arguably favor the latter. We leave this evaluation to future work.

VI. Related Work

There is currently very limited published research specifically addressing malware attacks on SGX, and to our knowledge none addressing the context we refer to in our work. The research presented in [21] is probably the closest approach to a malware targeting the supply chain infrastructure. The situation considered is of a malware code payload that is encrypted, downloaded, decrypted and executed in a secure enclave that was previously attested as legit. To initiate this sequence of steps, the attacker requires a remote bootstrap program that is used to build the initial enclave, attest it, and facilitate the exchange of keys with the attacker for encrypting the malware code. The decrypted malware code running inside the enclave can subsequently receive other instructions or input from the attacker. A use case also discusses the possibility, in the same manner, to also execute *stalling* code inside the enclave. Such code is not malicious *per se*, but it is typically executed before the malware and used for hiding the following malicious behavior from analysis tools.

Most of the documented attacks that target SGX rely on cache exploits. In [22], the authors assume the inclusion of malware in a malicious enclave co-located with a victim enclave. The malware performs a *Prime+Probe* cache side-channel attack through which it is able to recover RSA keys used in the victim enclave. The attacker uses the API provided by the victim to trigger signature computation and has to locate the victim's cache sets that contain the secret-dependent data of the leaked key. The primary purpose of hiding the malware inside an enclave is to conceal the malicious code, leveraging the SGX protection features to avoid detection. However, the attack does not target effectively infecting or corrupting the enclave where the malware resides, which is our case.

Another work [23] also profits from SGX isolation to stealthily operate by leveraging Intel's transactional synchronization extensions (TSX) memory-disclosure primitive. They show how to effectively bypass the host application interface and execute arbitrary system calls via return-oriented programming (ROP), without collaboration from untrusted code. Different from us, they do not target sensitive data operated by enclaves, but rather at hijacking the infected machine for subverted uses.

The authors of [24] present another attack on SGX that develops on the Prime+Probe technique of recovering information from the cache, such as an RSA key. In this case the attack isolates the core used by the victim enclave from other processes to minimize the noise in the side channel. Another improvement of the attack is uninterrupted execution by configuring the interrupt controller to not deliver interrupts to the attack core. If it is allowed on this core to receive interrupts, this could be used to deflect side channel attacks. The attack also relies on Intel performance monitoring counters (PMC) for monitoring cache evictions and also monitors the frequency in order to not miss victim accesses to the cache. In [25], the authors also describe an attack that uses CPU pinning and Intel PMC in a Prime+Probe approach. The attack retrieves cache information that leads to an AES key leak. For using these mechanisms the above attacks assume full control over the operating system where the victim enclave is run. In comparison, the attack in our scenario could be executed also under more restrictive conditions, but again the context is different since the referenced attacks do not effectively attempt to corrupt an enclave.

The recent Foreshadow attack [26] again targets CPU cache leaks, but adopts a different mechanism exploiting a speculative execution bug. This consists in an unauthorized memory access in transient out-of-order instructions, which can be used before rollback to retrieve confidential data, in a similar manner to the Meltdown attack [27]. Another recently-published side-channel attack [28] exploits contention on simultaneous multithreading (SMT) and code-reuse. They demonstrate their attack by finding gadgets from *glibc* to execute code that ultimately leak plaintext data from OpenSSL.

In [29], the authors describe an attack that violates the enclave integrity with the purpose of triggering a processor lockdown. The attack relies on the Rowhammer approach for flipping bits in the EPC memory region, which leads to the DoS effect. This is achieved by executing a code snippet inside the enclave that has to find conflicting row addresses in the same memory bank of the EPC, which is required to run the Rowhammer routine. The code snippet is supposed to be executed inside a malicious enclave that will be downloaded on a victim machine. Our attack scenario opens the possibility to corrupt legit enclaves with custom malicious code, which could also be such DoS triggering routines injected during the signing process, before attestation.

VII. Conclusion

We have presented a novel attack in the area of supply chain malware, with a specific target on protection measures that involve the use of SGX. We provided a practical use case for our attack methodology, which is able to successfully extract sensitive data from the secure enclave space. This use case is generic enough to be applied to multiple cases of enclaves. A malicious entity who has knowledge of a particular enclave functionality can leverage the attack scenario for more specific attacks that can be even more disabling, e.g., changing some particular behavior of the enclave code. The flexibility of the attack scenario, which requires essentially just a window of opportunity between the building and the signing of an enclave, makes it quite problematic.

Fortunately, some basic mitigation mechanisms are relatively easy to enforce, as discussed in Section V. We have shown that protection via compiling and securely signing the binary within a dedicated SGX enclave is a feasible practical option. Also, we believe some other more advanced mitigation options involving cryptographic techniques, such as verifiable secret sharing, could be explored. It is debatable, however, how well these schemes would perform in practice. As future work, we consider evaluating and comparing such mitigation mechanisms.

Acknowledgements

Some of the activities that contributed to this work were funded by the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 692178.

References

[1] R. Shaw, *Software supply chain attacks - NIST software and supply chain assurance winter forum 2017*, Accessed on 29/04/2019, 2017. [Online]. Available: https://csrc.nist.gov/csrc/media/projects/supply-chain-risk-management/documents/ssca/2017-winter/ncsc_placemat.pdf.

[2] Malwarebytes Labs, *Supply chain attack definition*, Accessed on 29/04/2019. [Online]. Available: https://blog.malwarebytes.com/glossary/supply-chain-attack/.

[3] (). CCleaner. Accessed on 29/04/2019, [Online]. Available: https://www.ccleaner.com/.

[4] Avast. (2018). New investigations into the CCleaner incident point to a possible third stage that had keylogger capacities. Accessed on 29/04/2019, [Online]. Available: https://blog.avast.com/new-investigations-in-ccleaner-incident-point-to-a-possible-third-stage-that-had-keylogger-capacities.

[5] Kaspersky Lab. (2017). ShadowPad - how attackers hide backdoor in software. Accessed on 29/04/2019, [Online]. Available: https://www.kaspersky.com/about/press-releases/2017_shadowpad-how-attackers-hide-backdoor-in-software-used-by-hundreds-of-large-companies-around-the-world.
[6] A. Cherepanov. (2017). Analysis of telebots cunning backdoor. Accessed on 29/04/2019, [Online]. Available: https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor/.
[7] V. Costan and S. Devadas, *Intel SGX explained*, Cryptology ePrint Archive, Report 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.
[8] Intel. (2018). Enclave signing key management. Accessed on 29/04/2019, [Online]. Available: https://software.intel.com/en-us/sgx-sdk-dev-reference-enclave-signing-key-management.
[9] ——, (2018). Request a commercial license. Accessed on 29/04/2019, [Online]. Available: https://software.intel.com/en-us/sgx/commercial-use-license-request.
[10] P. Fautrero *et al.* (2016). BeaEngine. Accessed on 29/04/2019, [Online]. Available: https://github.com/BeaEngine/beaengine.
[11] Intel. (2018). Code sample: Intel software guard extensions remote attestation end-to-end example. Accessed on 29/04/2019, [Online]. Available: https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.
[12] ——, (2018). The enclave signing tool. Accessed on 29/04/2019, [Online]. Available: https://software.intel.com/en-us/sgx-sdk-dev-reference-the-enclave-signing-tool.
[13] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *The 12th European Conference on Computer Systems (EuroSys '17)*, ACM, 2017, pp. 238–253. DOI: 10.1145/3064176.3064219.
[14] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP '16)*, Seoul, Republic of Korea: ACM, 2016, 10:1–10:9. DOI: 10.1145/2948618.2954331.
[15] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*, Atlanta, USA, 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43.
[16] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: SGX-based high performance BFT," in *The 12th European Conference on Computer Systems (EuroSys '17)*, ACM, 2017, pp. 222–237. DOI: 10.1145/3064176.3064213.
[17] F. Bellard *et al.* (2018). TinyCC. Accessed on 29/04/2019, [Online]. Available: https://github.com/TinyCC/tinycc.
[18] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni, and C. Fetzer, "SGX-aware container orchestration for heterogeneous clusters," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS '18)*, 2018, pp. 730–741. DOI: 10.1109/ICDCS.2018.00076.
[19] Intel, *Intel software guard extensions SDK developer reference for Linux OS*, version 2.5, 2019. [Online]. Available: https://download.01.org/intel-sgx/linux-2.5/docs/Intel_SGX_Developer_Reference_Linux_2.5_Open_Source.pdf.
[20] R. Pires, M. Pasin, P. Felber, and C. Fetzer, "Secure content-based routing using Intel software guard extensions," in *Proceedings of the 17th International Middleware Conference (Middleware '16)*, Trento, Italy: ACM, 2016, 10:1–10:10. DOI: 10.1145/2988336.2988346.
[21] J. van Prooijen, "The design of malware on modern hardware: Malware inside Intel SGX enclaves," University of Amsterdam, Tech. Rep., 2016. [Online]. Available: https://www.os3.nl/_media/2015-2016/courses/rp2/p89_report.pdf.
[22] 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*, M. Polychronakis and M. Meier, Eds., Springer International Publishing, 2017, pp. 3–24. DOI: 10.1007/978-3-319-60876-1_1.
[23] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with Intel SGX," *CoRR*, vol. abs/1902.03256, 2019. [Online]. Available: http://arxiv.org/abs/1902.03256.
[24] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, Canada, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.
[25] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel SGX," in *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*, ACM, 2017, 2:1–2:6. DOI: 10.1145/3065913.3065915.
[26] 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 *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, USA, 2018, pp. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.
[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, USA, 2018. [Online]. Available: https://www.usenix.org/node/217479.
[28] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," *CoRR*, vol. abs/1903.01843, 2019. [Online]. Available: http://arxiv.org/abs/1903.01843.
[29] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-bomb: Locking down the processor via Rowhammer attack," in *2nd Workshop on System Software for Trusted Execution (SysTEX'17)*, ACM, 2017. DOI: 10.1145/3152701.3152709.