University of Amsterdam
System & Network Engineering
Research Project 2

The Design of Malware on Modern Hardware

Malware Inside Intel SGX Enclaves

Jeroen van Prooijen
{jprooijen}@os3.nl

2nd August 2016

Abstract

Intel Software Guard Extensions (SGX) is a hardware technology that provides a secure execution environment for applications. When code is executed inside the SGX environment nobody can access the code. This means no kernel, hypervisor, operating system or administrator is able to modify or view the code. Normally executing software inside a trusted environment is a desired property, but what if malware is being executed using SGX technology? In this report, we try to answer the following question: "What would be the consequence if malware writers use the features of SGX?" Here, we show two proof of concepts in which malware benefits from SGX. The first proof of concept shows the remote attestation process to guarantee authenticity before downloading an encrypted payload. The second proof of concept implements a stalling mechanism, to delay the malicious behavior of malware. As a result, the only remaining way to analyze and mitigate SGX malware is system call heuristics.

Acknowledgements

I would like to sincerely thank my supervisors Kaveh Razavi and Marc X. Makkes who gave me the opportunity to do this research and guide me during the process. The brainstorm sessions and feedback were valuable to me.

I would also like to thank The Systems and Network Security Group of the Vrije Universiteit Amsterdam, especially Victor van der Veen, Cristiano Giuffrida, Dennis Andriesse and Radhesh Krishnan K. for their help and feedback.

Contents

1 Introduction

Intel SGX [1] is an extension to the Instructions Set Architecture (ISA) that includes special instructions regarding the interaction with memory. SGX aims to guarantee confidentiality and integrity on the execution of code by separating the memory on the hardware level and implementing special semantics for accessing the memory. According to Intel's specification no kernel, hypervisor or other program can access the code. Therefore, SGX will act as a secure vault for executing code. This secure vault is named *enclave*. In addition, SGX also provides a method for software attestation to ensure that the enclave is setup correctly and placed on an authentic Intel platform. The attestation process can be executed both locally and remotely. A SGX-enabled program can start an enclave running on the same privilege ring as itself. However, running in the same privilege ring as the application, it is not possible for higher privileged rings to access the enclave due to the hardware level separation and detection enforced by SGX.

This research describes the consequences when malware is starting to use SGX. Use of SGX can offer malware developers a way to protect their malicious programs against analysis. When only the attacker can access the malware running inside an SGX enclave this can make analysis of the malware more troublesome. The features that SGX provides make it impossible to do static or dynamic code analysis. Malware analysts use static and dynamic code analysis to study the behavior of malware. Therefore, studying SGX malware needs a different approach than non-SGX malware.

Two proof of concepts were developed to illustrate the possibilities for malware to use SGX. At first a program was developed that uses the remote attestation process to ensure that an enclave is setup correctly. During the attestation process a bootstrap program will be executed outside an enclave. The bootstrap program is susceptible for analysis up to the moment when an attestation is done successfully. A successful remote attestation results in a secure environment for the attacker to sent malicious code to. This could be a valid use case for malware writers to use SGX. The second proof of concept is a program that implements stalling code inside an enclave. Malware writers use stalling code to keep the malware in a dormant state. When the malware is being analyzed the stalling code will withhold the decryption and execution of its malicious code. Using stalling code will not reveal the malicious code and therefore make it harder to analyze. Executing stalling code inside an enclave could be useful from an attacker's perspective. Both proof of concepts were researched to determine possible analysis methods. The code executed inside an enclave can be analyzed using system calls. Monitoring enclaves and generating system call patterns can help in SGX malware analysis.

In summary, we make the following contributions:

- An initial exploration on the consequences of SGX-enabled malware.
- We developed two small proof of concept programs which illustrate the use of SGX in malware.
- We tested these proof of concept programs to determine which analysis methods would still be useful.

This report is organized as follows: Chapter 2 includes related work. Chapter 3 provides background related to this research. In chapter 4 the research questions are defined together with the design and analysis of two example use cases for malware. Chapter 5 discusses some remaining issues and future work. Chapter 6 concludes this research.

2 Related work

In the past years research is done on shielding applications [2, 3] and providing secure computations on untrusted platforms [4–7]. A different approach would be executing computations on encrypted data, known as *fully homomorphic encryption* [8], or use *Controlled Physical Uncloneable Functions (cPUFs)* as a method to certify and prove execution [9]. In addition, different hardware vendors have developed products to aid in providing secure platforms. ARM has developed ARM TrustZone [10]. AMD has embedded the ARM technology and adopted it as Platform Security Processor (PSP) [11, 12] and Intel has developed TXT [13, 14]. There is also an international standard for securing hardware with the use of a Trusted Platform Module (TPM) [15]. Software attestation is possible in multiple ways [16–18], Intel has provided an open source project called "The Open Attestation (OAT) Project" [19]. In SGX some of the features mentioned above are combined to provide a *Trusted Execution Environment (TEE)* and software attestation.

Earlier SGX specific research, named Haven, has shown the possibility to execute unmodified legacy applications inside SGX [20]. Haven uses the Drawbridge project [21] and a library operating system (LibOS). Although, Haven code was executed inside an enclave it was still possible to analyze it. Xu, Cui and Peinado [22] published a side-channel attack that uses page faults to extract large amounts of sensitive information. The vulnerabilities lie in the implementation as Intel did not intend SGX to be used this way. Another SGX research allows the execution of a MapReduce computation while maintaining the confidentiality and integrity by executing the computation inside SGX [23]. Costan and Devadas [24] recently published a detailed summary of the publicly available information on SGX.

Malware developers use different techniques such that malware evades analysis [25, 26]. Communication in botnets is getting more resilient [27] and has moved to peer-to-peer communication models [28, 29]. Yan, Chen and Eidenbenz researched the possibilities of making peer-to-peer botnets resilient to enumeration [30]. Yan, Ha and Eidenbenz did some research on developing a peer-to-peer botnet that is more resilient to pollution [31]. Malware also focuses on the use of low level hardware access, by using Direct Memory Access (DMA) [32] and by hiding itself in the firmware of hard drive controllers [33]. SGX is a hardware technology that can execute code without being able to view or access the code. In addition SGX technology looks promising for applications and is actively researched. Malware developers are making malware more resilient to analysis. Malware developers could combine the SGX technology to make malware more resilient to analysis. This research focuses on the consequences of SGX malware in the design and analysis of malware.

3 Background

This chapter will explain the fundamentals needed to understand the working of SGX. First, in 3.1 we will briefly describe SGX itself. Then, in 3.2 we will explain software attestation which is used by SGX to verify if an enclave is authentic. In 3.3 the needed steps to develop SGX-enabled programs are described. Section 3.4 describes a benign use case of SGX in cloud computing. In section 3.5 some background information is given on botnet analysis. To conclude, in section 3.6 we explain what stalling code is and why malware developers use stalling code.

3.1 Intel SGX

SGX provides a hardware based trusted execution environment. SGX provides confidentiality and integrity of the code being executed. The trusted execution environment can be described as a vault to execute code in. Intel named this vault an *enclave*. Intel has implemented SGX by extending the existing instruction set to allow for special instructions regarding these enclaves. These include but are not limited to the creation and deletion of enclaves and entering and exiting the enclave. When the Central Processing Unit (CPU) has entered an enclave it is switched to *enclave mode*. Only then can the CPU access, decrypt and execute the code belonging to that enclave.

Executing an application with SGX support will use the extended instruction set to initiate an enclave. After initiation and attestation the enclave can be trusted with secret data. The application can still communicate with the code executing inside the enclave using a SGX specific application design. The enclave will execute in the same privilege ring as the application it is created by. When the code of an enclave needs to transfer from CPU cache to memory it is encrypted before being placed into memory. Snooping on the memory bus or memory itself will only result in encrypted data.

3.2 Attestation

To guarantee that an enclave is setup correctly *software attestation* is used. This attestation process provides a way to verify if an enclave passes the initial integrity check and is initiated on an authentic Intel platform. A verifying party can make sure that the enclave is set up as meant to be before confidential code is being loaded inside the enclave. This attestation is a key part of SGX as it builds a trust relation with the enclave placed on a possibly compromised system. Intel recommends to use attestation when setting up an enclave [34, chap. 1.3].

Intel has developed two ways of attestation: *local attestation* and *remote attestation*. Local attestation provides mutual authentication between two enclaves. Remote attestation can be used for verifying enclaves from a remote location. A key role in the attestation process is the use of two registers called *MRENCLAVE* and *MRSIGNER*. The MRENCLAVE is a SHA-256 digest of the enclave code, data and stack. The MRSIGNER identifies the builder of the enclave, most likely the developer or its company. An RSA signed certificate, with the supposed initial values of the MRENCLAVE and MRSIGNER, can be used to verify if these values correspond with the actual registers of the enclave. If so, the enclave is set up as it is supposed to and can be used to load and execute confidential code.

For this research the remote attestation is of more importance than the local attestation. Here, we will only describe the details of remote attestation. For more information on local attestation we point to the Intel documentation [35, 36].

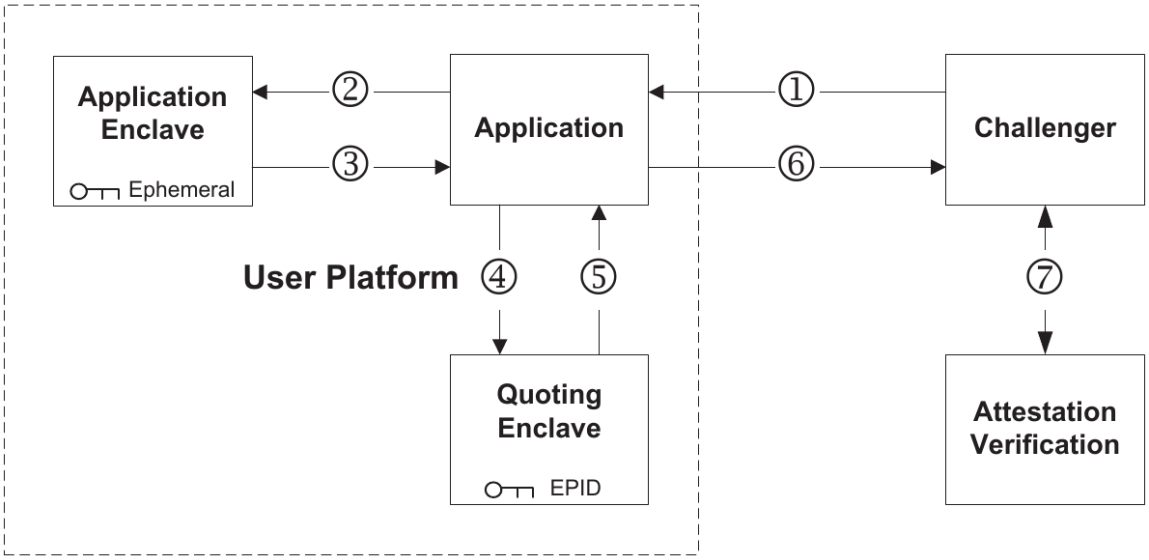

Figure 2: Remote attestation [35].

In figure 2 the remote attestation is described, starting from a third party called *Challenger*. Most likely this third party would be the software vendor. The SGX-enabled application would be executed on a remote machine, illustrated with the dotted line in figure 2. The description below is partly taken from Intel's documentation [35].

1. The challenger does an attestation request to the application.
2. The application forwards the request to its enclave.
3. The enclave is generating a *REPORT* [37, p. 22] using the *EREPORT* instruction [37, p. 143]. This REPORT resembles the enclave including the measurement registers. The REPORT is being sent to the application. The enclave can also start an asymmetric key exchange in the REPORT such that after the attestation the challenger can communicate confidential information back to the enclave directly [36, chap. 6] [35, chap. 3.2.3].
4. The REPORT is being forwarded to the Quoting Enclave.
5. The Quoting Enclave is a special enclave Intel has developed and is always available on an SGX-enabled platform. The Quoting Enclave can verify the REPORT and can sign the result using the EPID key. This results in a *QUOTE*. The QUOTE resembles a verification done by the CPU itself.
6. The QUOTE is being sent to the Challenger.
7. The Challenger can then verify if the application enclave is setup correctly by checking the QUOTE. The Challenger could use the EPID public key certificate or use an Intel provided Attestation Verification Service.

REPORT and QUOTE are data structures that describe either the enclave or the result of the check executed by the Quoting Enclave. The REPORT and QUOTE structures are documented by Intel [37–39]. For a basic understanding the exact structure format can be left out, but keep in mind that the details are important to proof the authenticity and integrity of an enclave.

3.3 Development

Developing SGX-enabled programs has a different development cycle compared to non-SGX programs. There are a five important steps during the process of developing applications that are different compared to the development of non-SGX applications. Documentation on using the Software Development Kit (SDK) under Windows, can be found on Intel's website [40]. This section describes the additional steps that developers need to take to develop SGX-enabled programs:

- Writing enclave code is similar to writing normal code.
- To make use of the enclave code an interface needs to be defined which describes the functions that can be called from the untrusted part of the program to the trusted part of the program inside the enclave. This definition plays a key role as it determines the communication channel of an enclave. Mistakes in this definition can lead to a compromise or memory leakage of enclave code [36, chap.

2]. The language used for describing this interface is called Enclave Definition Language (EDL) [41].
- Then the enclave code and the EDL can be imported in an application. After importing the enclave code it is possible for the untrusted code to call the functions that are described in the EDL and provided by the enclave code.
- After compiling the program, the enclave code is placed inside a separate enclave file.
- The enclave file needs to be signed by the developer, which is required by the SGX architecture [36, chap. 4.1].

3.4 Benign Use Case

Cloud-based computation on sensitive data needs to be implemented such that the confidentiality of the data is preserved. SGX could be used for this purpose. A benign use case would be the execution of a distributed MapReduce instruction in a cloud-based environment. An example implementation of this instruction is *Verifiable Confidential Cloud Computing (VC3)* [23]. VC3 uses SGX for executing a secure MapReduce instruction. The process of executing the MapReduce instruction is illustrated in figure 3.

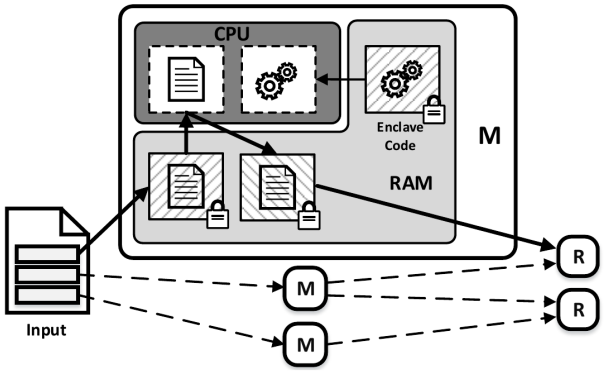

Figure 3: High-level concept of the VC3 implementation of the MapReduce job [23, chap. III].

The encrypted input is uploaded to the memory of a remote machine. Once an enclave is initiated and attested the input is placed in the enclave and decrypted. After decryption, the entire MapReduce instruction is executed in the protected environment of the enclave. The result of the MapReduce instruction is encrypted and placed back in memory.

3.5 Botnet Analysis

Botnet research on non-SGX malware could first focus on *intelligence gathering* and later switch to a *disruption and destruction* of the botnet [27, 29]. A researcher has possibilities to interfere with the code by changing or reverse engineering the program. Another possibility is to interfere with the structure of the botnet by changing the code to delete edges or inject peers into a peer-to-peer botnet. Having the ability to interfere with the code is of importance when researching botnet related malware.

3.6 Malware Stalling Behavior

A desired feature for malware is that it is not detected when searched for. There are multiple ways of trying to evade analysis. One of the techniques used is called *stalling*. Most often this is combined with an encrypted payload which is only decrypted after the stalling code is done. When the malware is being analyzed stalling code keeps executing a certain loop which does nothing harmful but only generates bogus data to create a haystack of analysis information. Furthermore analysis tools are generally capable of running a certain amount of time. Having malware stall long enough will keep it's malicious intent hidden. As a reaction to stalling code, malware researchers have developed ways to evade or skip stalling code. One of the approaches is called *Hasten* [42]. Hasten uses system call heuristics and a Control Flow Graph (CFG). This CFG is generated by conducting dynamic analysis on the memory that's being used by the malware. Hasten also implements a way of interfering with the code such that stalling loops can be skipped.

4 Research

This chapter describes the approach of this research. At first, 4.1 describes the research questions which were defined. Section 4.2 describes the platform we've used to develop the proof of concepts. Section 4.3 describes two possible use cases for malware to leverage the features of SGX. As an example we used the attestation process and developed a way to execute stalling code inside an enclave. The attestation and stalling examples are described respectively in 4.4 and 4.5.

4.1 Research Questions

Assuming that SGX is going to be used widely this would offer malware writers a possibility to harden their malware. The main reason to investigate the possibilities for malware to use SGX is the changes it may cause for malware analysis. The fact that SGX provides confidentiality and integrity guarantees over the code being executed inside an enclave can make analyzing this code hard. This research aims to get more acquainted with the possibilities for malware to use SGX and the possibilities for researchers to analyze SGX-enabled programs. In this report we try to answer the following research question:

"What are the consequences of SGX-enabled malware?"

In order to answer the main research question we defined subsidiary questions. Malware writers are using different techniques to make hardware more resilient or harder to analyze. The first subsidiary question is:

"How could malware benefit from using SGX?"

Some adjustments need to be made in order to use the features that SGX offers. The phases an SGX application has are different compared to a non-SGX application. This leads to our second subsidiary question:

"What adjustments in the design of malware need to be made in order to make use of SGX?"

SGX is developed to achieve confidentiality and integrity on the code executed in the protected environment. When researchers study the behavior of malware that executes inside SGX this would most likely cause changes in the approach of malware analysis. Our third subsidiary question would be:

"What methods for analyzing malware still work and which are obsolete?"

4.2 Development Platform

At the start of this research Intel had not made their SGX SDK publicly available yet. Only a few research groups had access to the official SGX code and platform. Jain et al. [43] have developed a platform, called OpenSGX, meant for researchers to study the capabilities and features offered by SGX. During this research OpenSGX was used to develop and test the proof of concept programs of section 4.3. Although, OpenSGX does not offer the same level of protection as SGX, it is still representative for the high-level separation that SGX offers.

4.3 Use Cases

There are multiple use cases one can think of when developing code for SGX enclaves. To scope the research we decided to turn the two following use cases into working proof of concepts:

1. For malware purposes we can use the **attestation** process to verify a remote platform and only upload the malicious payload when attestation is successful. SGX will guarantee that the enclave is authentic and trusted. Therefore, malware developers will have a trusted execution environment.
2. Executing **stalling code** inside an enclave will keep the stalling code protected against modification. Therefore, analysis of stalling code executed inside SGX will be different compared to normal malware analysis.

The reason for the attestation example is that remote attestation is a unique feature of SGX and has interesting possibilities from a malware perspective. The reason for the stalling code example is that it illustrates the challenges that malware analysts are going to face.

4.4 Attestation

This section describes our first proof of concept. We use the remote attestation process to illustrate the use for malware developers to harden their malware? In 4.4.1 we describe our design. Section 4.4.2 describes the analysis done on the developed proof of concept. Using the remote attestation feature available in SGX provides a way to verify that an enclave is initialized on an authentic Intel platform. After attestation the malware developer is assured of a trusted execution environment to execute its malware in. For analysis this results in a approach focused on system call heuristics.

4.4.1 Design

The initial application is placed on a remote platform without providing confidentiality or authenticity. Initializing the enclave is done in plain sight. No secrets can be provided before the initial set up of the enclave. Only once initialized and attested successfully one knows that the platform is authentic and can transfer secrets to an enclave [24, chap. 6.6.1]. This approach of writing a program will result in a generic bootstrap program which can have any use case.

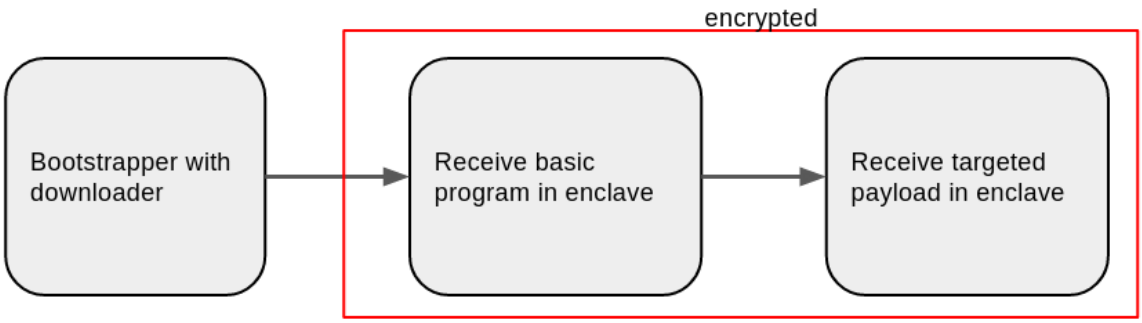

Figure 4: An abstract design of the different phases that malware can be in.

Figure 4 illustrates the phases of possible SGX malware. A generic bootstrap application with downloading capabilities is placed on a remote platform. The bootstrap application has no payload which makes the use of obfuscation and different packers unnecessary. When the initial bootstrap program is started it will setup an enclave. This enclave is then verified and attested using the remote attestation process. At the same time it will do a key exchange with the malware developer. After a successful attestation an encrypted payload can be sent directly to the enclave. This payload can contain basic malware functionality. After receiving, this code is decrypted in the enclave and executed inside the enclave waiting for further instructions, be it a command, targeted payload or update. Using a generic bootstrap program could also be used to evade anti-virus systems [24, chap. 5.9.5].

Here, a proof of concept program was developed to test the possibilities of using SGX and remote attestation for malware purposes. A simple bootstrap program was used to initialize and enter an SGX enclave. Listing 1 illustrates the code that is used for the bootstrap part of the program.

```
1  int main(int argc, char **argv)
2  {
3      tcs_t *tcs = prepare_enclave(argc, argv);
4      void (*aep)() = exception_handler;
5
6      enter_enclave(tcs, aep, argc, argv);
7
8      return 0;
9  }
```

Listing 1: Basic bootstrap code.

Inside the enclave we use the attestation functions, as shown in listing 2. When the attestation is successful we know that an enclave is setup securely and we can provide our secrets to that enclave.

```
void enclave_main()
{
    int challenger_port, ret;
    char *ff_domain = "20ajf412.biz";
    char *ff_payload = "49fhsb24.biz";
    challenger_port = 8024;

    ret = sgx_remote_attest_target(ff_domain, challenger_port, QUOTE_PORT);
    if(ret == 1) {
        puts("Remote Attestation Success!");
        get_payload(ff_payload, 443);
    } else {
        puts("Remote Attestation Fail!");
    }
    sgx_exit(NULL);
}
```

Listing 2: Attestation and download code.

4.4.2 Analysis

As mentioned earlier, no secrets should be provided before a successful attestation is done. As the bootstrap code is not encrypted this means that domain names or IP addresses can be extracted from the program. From a malware perspective this means that it is still necessary to use fast flux techniques [44] or similar to obfuscate the communication endpoints.

Our enclave code is running in the same privilege ring as the host application, being the bootstrap program. In this case the privilege ring would be ring 3. This means that for every IO operation a system call needs to be executed. When executing the attestation example from listing 2 The GNU Project Debugger (GDB) [45] can be used to debug the code. Placing a monitor on the *connect* system call allows for intercepting the IP address of the host that the program is trying to communicate with.

```
Catchpoint 1 (call to syscall connect), 0x07fefc34c57f0 in __connect_nocancel ()
(gdb) n
connect () at ../sysdeps/unix/syscall-template.S:86
(gdb) n
do_connect (sockfd=6, target_addr=2155874016, addrlen=16)
(gdb) x/2xb target_addr
0x808006e0:     0x02    0x00
(gdb) x/2xb target_addr+2
0x808006e2:     0x1f    0x58
(gdb) x/4xb target_addr+4
0x808006e4:     0x7f    0x00    0x00    0x01
```

Listing 3: GDB output of *connect* system call.

The domain in listing 2 is changed to use the localhost address to connect to. The connect system call takes three variables of which the second one is a *const struct sockaddr pointer*. This struct includes the port and IP address [46, 47]. When debugging the system call the port and IP address can be retrieved by investigating this struct, named *target_addr* in listing 3. In table 1 the values that were found are converted to a more readable format.

HEX value	readable value	description
0x1f58	8024	port
0x7f000001	127.0.0.1	IP address

Table 1: Hexadecimal and readable values.

Capturing the hexadecimal values illustrates the possibility for using system calls in analyzing SGX programs. In this example the system call is setting up an connection. When an encrypted channel is established the monitoring of system calls relating to the communication would result in encrypted data. Assuming the encryption is strong enough this would be less interesting. From an attacker's perspective this means that the code itself remains hidden and only allows for system call heuristics. Changing the system call pattern will evade known pattern detection.

4.5 Stalling Code

This section describes our second proof of concept. This proof of concept illustrates the use of SGX to implement stalling behavior inside an SGX enclave. Section 4.5.1 describes our design. Section 4.5.2 describes the analysis done on the proof of concept. Researchers studied the stalling behavior in malware, as a result there are possibilities to skip stalling loops. The availability of SGX gives malware developers new possibilities to execute stalling code inside the protected environment of SGX. Using functions that don't create system calls can make the malware hard to analyze.

4.5.1 Design

During the analysis of the attestation example in 4.4.2 we've already determined that system calls can be viewed when interaction occurs with higher privileged rings. This means that for a *sleep* function the corresponding system call can also be monitored. The sleep function can be evaded during analysis by blocking the system call in a way that it is being skipped. To harden the sleep function we need to verify if an actual sleep has occurred. By verifying the sleep function we can make sure that no stalling code is being skipped.

In the Intel architecture there is a instruction called Read Time-Stamp Counter (RDTSC) [48, p. 307]. This RDTSC instruction reads the time-stamp counter which returns the number of clock ticks passed since the last reset of the CPU. However, this seems not exact enough to measure time precisely. The exact behavior of the time-stamp counter is out of scope in this research. Important to notice is that executing the RDTSC instruction will not cause a system call.

```
1  static inline uint64_t get_cycles_x64()
2  {
3      uint64_t lo, hi;
4      __asm volatile ("rdtscp" : "=a"(lo) , "=d"(hi));
5      return (hi<<32)|lo;
6  }
7
8  void enclave_main()
9  {
10     uint64_t c1, c2, diff;
11     float div, time;
12     int count=0;
13
14     for (int i=0; i<600; i++){
15       c1 = get_cycles_x64();
16       sleep(1);
17       c2 = get_cycles_x64();
18       diff = c2-c1;
19       if (diff > 500000000){
20         count++;
21       } else {
22         sgx_exit(NULL);
23       }
24     }
25     if (count == 600)
26       execute_payload();
27 }
```

Listing 4: Stalling enclave code.

In this proof of concept we still use the bootstrap code from listing 1 to enter the SGX enclave. We assume that a remote attestation was successful and that the code in listing 4 is downloaded in the enclave after doing attestation. Therefore, the confidentiality and integrity of the stalling code are preserved. The instruction RDTSCP in listing 4 is slightly different from the RDTSC instruction. This derivative of the RDTSC instruction is executed serialized [48, p. 547], which guarantees the correct order of executing the instruction just before and after our *sleep* function on line 15-17.

When comparing both variables *c1* and *c2* we can verify that clock cycles have passed. On line 19 we check if half a billion clock cycles passed by. The exact value of this number is not important as long as it is high enough to determine some amount of time has passed. We will discuss the reason of this value in section 4.5.2.

4.5.2 Analysis

Just by looking at the code of listing 4 we can already point out which lines would be interesting to skip when analyzing this code. If we skip the *sleep* function and the comparison on line 25, the fictive payload would be executed and therefore accessible for analysis.

```
(gdb) catch syscall nanosleep
Catchpoint 1 (syscall 'nanosleep' [35])
(gdb) c
Continuing.

Catchpoint 1 (call to syscall nanosleep), x7f8a69d75c40 in __nanosleep_nocancel()
(gdb) n
do_syscall (cpu_env=0x5591a152a1e0, num=35, arg1=274886290560, arg2=274886290560, arg3
    ↪ =2155873280, arg4=1845, arg5=0, arg6=1, arg7=0, arg8=0)
(gdb) x arg1
0x40007fec80:    0x00000001
(gdb)
```

Listing 5: GDB output of *nanosleep* system call.

The system call that is done when executing the *sleep* function can be detected, as shown in listing 5. The comparison on line 25 of listing 4 however can not be detected as it does not cause a system call to occur. This means that skipping the sleep will result in a *count* value not being equal to 600. Even if one manages to change the memory inside the enclave this will trigger the integrity checks and will stop the enclave and therefore stop the analysis.

Since the RDTSCP instruction is not precise some measurements were done to determine the variance of time measurement using the RDTSCP instruction. We tested the code of listing 4 without OpenSGX support. The reason why we tested the code outside OpenSGX is that SGX is an ISA extension and OpenSGX uses QEMU to emulate the SGX instructions. Testing the code with OpenSGX needs some basic libc functions to log timing results to a file. OpenSGX didn't have these libc functions implemented. Implementing the needed functions in OpenSGX was possible but would create more overhead and therefore a deviation in the timing compared to non-emulated instructions.

Measuring the timing of a *sleep* function was done by calculating the difference of the RDTSCP before and after the *sleep* function was called. The values before and after, respectively *c1* and *c2*, would then be divided by the base clock speed of the CPU.

$$\frac{c2 - c1}{CPU\ cycles\ per\ sec.} = sleeptime\ in\ sec. \tag{1}$$

The boxplot of figure 5 illustrates the amount of time in seconds that was calculated using the CPU speed and the difference between two RDTSCP instructions. The tests were conducted on a laptop with an Intel i7-3520m processor and repeated 5000 times. The CPU was set with the *constant_tsc* option enabled. The base CPU speed of this processor is 2.9 GHz. To summarize we can say that the number of clock cycles for a 1-second sleep needs to be around 2.9 billion clock cycles. The little circles in figures 5 and 6 represent the outliers.

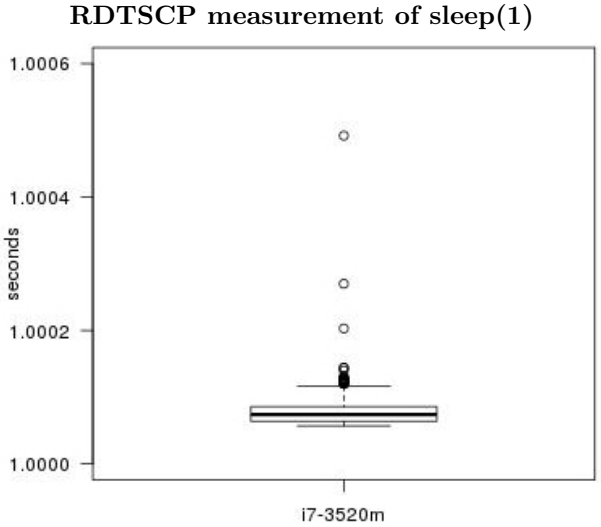

Figure 5: Boxplot of the results when calculating the passed time using the RDTSCP instruction before and after executing the *sleep(1) function*.

To measure the lower boundary, the stalling program was executed while suppressing the *sleep* function. The boxplot in figure 6 illustrates the results when the *sleep(1)* function is being skipped. We use the amount clock cycles instead of the calculated time since it makes the scale on the y-axis more readable. The amount of clock cycles that have passed can be calculated by measuring the difference of variable *c1* and *c2*. Figure 6 shows a median of around 275 clock cycles.

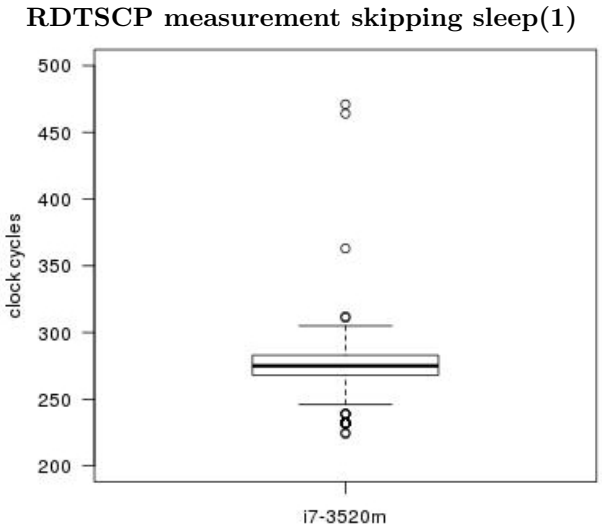

Figure 6: Boxplot of the results when calculating the amount of clock cycles passed between two RDTSCP instructions while skipping the *sleep(1)* function.

The results from our tests show that there is a variance when using the RDTSCP for calculating time. However, this approach is precise enough to detect if the *sleep* function is skipped or not. An estimation can be made for the number of clock cycles that need to pass for every 1-second sleep. Choosing a value of half a billion clock cycles seems sufficient to determine that a *sleep(1)* function is skipped or not. Intel CPUs with SGX support don't have a base clock speed lower than 1.6 billion cycles per second (1.6 GHz) [49]. Therefore, we can assume that using half a billion clock cycles will also work as expected on an SGX platform with a different clock speed. To conclude SGX can aid an attacker in hiding the malicious intent. The malware is able to stall the decryption and execution of the payload without the use of system calls while avoiding modification to the stalling code.

5 Discussion and Mitigations

This chapter will discuss the results of this research. At first, section 5.1 discusses the privilege ring that enclave code is running in. Section 5.2 describes the consequences for botnet research. Section 5.3 describes the limitation of our stalling example. Section 5.4 describes a possible mitigation for the stalling proof of concept. In section 5.5 generic mitigations are discussed. Section 5.6 discusses the ability of Intel to monitor attestation results. To conclude this chapter some future work is discussed in section 5.7.

5.1 Privilege Ring

Intel has made a design choice to let the enclaves run under the same privilege level as the application, being ring 3. The design of SGX makes it impossible for higher privileged rings to access the enclave. This design is the reason why enclave code can be shielded against malicious intent by for example the operating system. Running enclaves under the same privilege ring means that every operation related to IO, reading/writing files or network communications, need to be done using system calls.

5.2 Botnet Research

When malware developers are implementing SGX malware, botnet analysis will become harder. If malware uses the remote attestation provided by SGX this means that the malware authors can verify that the platform is based on genuine Intel hardware. The malicious payload will only be downloaded when the enclave is set up and attested for. Furthermore, the code is only available decrypted in the cache of the CPU, and changing it somehow during execution will trigger the integrity checks, stop the malware and empty the cache. The features of SGX take away the possibility to modify the malware running inside the SGX enclave. As a result the techniques to take over and take down a botnet that are available now will become obsolete.

5.3 Stalling Code Limitations

Hypervisors can emulate the RDTSC instruction [50, 51] and SGX is a hardware based technology. When enclaves are initialized by an application executing in a hypervisor this could have consequences for the reliability of the RDTSC instruction. At this moment no research is known regarding hypervisors initiating enclaves. How SGX is used by hypervisors still needs to be seen. For now there is no documentation about trustworthy timing sources available from within an SGX enclave.

5.4 Stalling Code Mitigation

The stalling code example from chapter 4.5 has proven to be effective. Code that can be identified with stalling behavior causes difficulties in analysis when executed inside an enclave. Due to the features provided by SGX there is no way to interrupt and modify the code executed in the enclave without breaking the process. To mitigate our stalling example Intel would need to make the RDTSC and related instructions not callable from within an SGX enclave. Making the RDTSC instruction not callable will disable the use of the RDTSC instruction for time measurement as shown in our proof of concept, documented in section 4.5.

5.5 Generic Mitigation

When looking at earlier research on system call heuristics [42] a conclusion can be drawn that only part of the Hasten approach is still usable. Doing system call heuristics and thereby building a signature of the malware can still be used. Building a CFG however is not possible anymore due to the fact that the code is protected by the confidentiality measures of SGX. These will take away the possibility to build a CFG to get a better understanding of the program flow. The same applies to interfering and modification of the code running inside an enclave. Skipping over stalling loops using the Hasten approach will not work in SGX malware. From a prevention perspective system call heuristics could be helpful as system calls from applications can be blocked and therefore malware inside enclaves can be blocked as well.

5.6 Intel Attestation Verification Service

During the remote attestation process a third party can verify the enclave. This third party will most likely be the software vendor that provides the SGX-enabled application. The third party can use Intel's Attestation and Verification Service to determine the integrity and authenticity of the platform where the enclave is setup. During this process there can be communication between the third party and Intel's Attestation and Verification Service. The fact that this communication happens also means that Intel has the ability to monitor the requests for this service and by whom the requests are made. Being able to monitor the attestation verification could be valuable when doing SGX malware analysis. Having a better understanding who is doing a remote attestation verification can add insight on the inner workings of SGX-enabled malware, especially botnet infrastructures. Intel has the ownership of the network traffic considering the verification service. Having this data places Intel in a position that it can decide who can do analysis on this data and who not. This would enforce their near-monopoly position on the market. Intel should use this information with integrity and responsibility as it can damage their reputation and the ability to do research.

5.7 Future Work

There are two topics that still remain to be researched. Section 5.7.1 describes the porting from OpenSGX to SGX. Section 5.7.2 describes improvements on the remote attestation proof of concept.

5.7.1 Intel SDK

At the end of this research Intel published their official SDK for Windows and Linux [52–55]. OpenSGX programs are not binary compatible with SGX. In order to execute the two example proof of concepts in SGX one needs to rewrite them. Interesting research can be porting the proof of concept code (OpenSGX) to SGX. One of the differences with the OpenSGX framework is that OpenSGX uses a technique called *Trampoline and Stub* which allows the implementation of wrappers around libc functions. Using Trampoline and Stub makes it possible to execute libc functions from within an enclave. When such function is invoked the trampoline makes sure that the function is getting called and the result is stored in a shared memory location before being passed to the enclave. This approach makes sure that there is a way of separating memory between the enclave and the 'untrusted' platform.

Intel uses a slightly different approach by defining the trusted and untrusted interface between the host process and the enclave using the EDL [41]. The tool that compiles the EDL file into source code is called Edger8r [56]. The exact workings and output of Edger8r can be interesting since that determines the way how the host process and the enclave can communicate. This communication interface plays a key role in the security of enclaves.

5.7.2 Remote Attestation

The remote attestation in OpenSGX has multiple dissimilarities compared to SGX. In SGX the REPORT is generated using a hardware instruction. OpenSGX doesn't have this instruction, but uses a configuration file with the same structure as the REPORT data structure [37, p. 22]. During this research a couple of bugs were found in the implementation of the attestation process in OpenSGX. For these a bug report was made [57, 58]. The issues that were found are pointing to incorrect parsing of the REPORT configuration file when the program is in *enclave mode*. The exact cause of the problem was not discovered during this research. Intel has made their SDK public which means it is now possible to use supported hardware and interact with the Attestation Server of Intel. Interesting research would be the investigation of the workings and limits of this Attestation Verification Service that Intel provides.

6 Conclusion

The consequence of SGX-enabled malware is that malware analysts can only use system call heuristics. Malware can benefit from the features provided by SGX. From a attacker's perspective SGX provides a Trusted Execution Environment (TEE). The attestation proof of concept shows that after doing a remote attestation the enclave can be proven authentic. This SGX feature allows malware developers to determine if an infected host can be provided with a confidential payload. An enclave provides confidentiality and integrity, this makes static code analysis obsolete. In our stalling proof of concept we showed that otherwise easily detectable stalling code can be hard to analyze when executed inside an enclave. Every SGX application needs to define the interface between the host application and the enclave. Malware writers need to adjust their code in order to make a generic loader that starts an enclave for their payload. Furthermore, the attestation process will cause changes in the design of malware. A malware designer also needs to think about the system calls that his code executes. Enclave code is executed on the same privilege ring as the application, being ring 3. For investigating SGX malware this would result in a method focused on monitoring system calls. Building different system call patterns can aid in the detection and identification of different malware payloads. Modifying malware code while executed inside an SGX enclave will not be possible due to the features SGX provides. Static analysis of the payload will become obsolete since the payload will only be decrypted inside an enclave.

References

[1] (2016). Intel Software Guard Extensions, [Online]. Available: https://software.intel.com/en-us/sgx (visited on 31/05/2016).

[2] L. Singaravelu, C. Pu, H. Härtig and C. Helmuth, 'Reducing TCB Complexity for Security-sensitive Applications: Three Case Studies', in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06, Leuven, Belgium: ACM, 2006, pp. 161–174, ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217951. [Online]. Available: http://doi.acm.org/10.1145/1217935.1217951.

[3] S. Kim, Y. Shin, J. Ha, T. Kim and D. Han, 'A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications', in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV, Philadelphia, PA, USA: ACM, 2015, 7:1–7:7, ISBN: 978-1-4503-4047-2. DOI: 10.1145/2834050.2834100. [Online]. Available: http://doi.acm.org/10.1145/2834050.2834100.

[4] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee and E. Witchel, 'InkTag: Secure Applications on an Untrusted Operating System', *SIGPLAN Not.*, vol. 48, no. 4, pp. 265–278, Mar. 2013, ISSN: 0362-1340. DOI: 10.1145/2499368.2451146. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451146.

[5] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig and A. Vasudevan, 'OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms', in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: ACM, 2013, pp. 13–24, ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516678. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516678.

[6] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz and D. Song, 'PHANTOM: Practical Oblivious Computation in a Secure Processor', in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: ACM, 2013, pp. 311–324, ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516692. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516692.

[7] J. Criswell, N. Dautenhahn and V. Adve, 'Virtual Ghost: Protecting Applications from Hostile Operating Systems', in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, Utah, USA: ACM, 2014, pp. 81–96, ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541986. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541986.

[8] C. Gentry, 'A Fully Homomorphic Encryption Scheme', PhD thesis, Stanford University, 2009. [Online]. Available: https://crypto.stanford.edu/craig.

[9] B. Škorić and M. X. Makkes, 'Flowchart Description of Security Primitives for Controlled Physical Unclonable Functions', *Int. J. Inf. Secur.*, vol. 9, no. 5, pp. 327–335, Oct. 2010, ISSN: 1615-5262. DOI: 10.1007/s10207-010-0113-2. [Online]. Available: http://dx.doi.org/10.1007/s10207-010-0113-2.

[10] ARM. (2016). ARM TrustZone, [Online]. Available: http://www.arm.com/products/processors/technologies/trustzone/ (visited on 18/07/2016).

[11] W. William. (7th May 2014). Platform Security Processor Protects Low Power APUs, [Online]. Available: http://electronicdesign.com/microprocessors/platform-security-processor-protects-low-power-apus (visited on 18/07/2016).

[12] AMD. (2016). AMD Secure Technology, [Online]. Available: https://www.amd.com/en-us/innovations/software-technologies/security (visited on 18/07/2016).

[13] Intel. (2016). Trusted Compute Pools with Intel Trusted Execution Technology, [Online]. Available: http://www.intel.com/txt (visited on 18/07/2016).

[14] ——, (21st Jan. 2015). Intel Trusted Execution Technology (Intel TXT) Enabling Guide, [Online]. Available: https://software.intel.com/en-us/articles/intel-trusted-execution-technology-intel-txt-enabling-guide (visited on 18/07/2016).

[15] Trusted Computing Group. (2016). TPM Main Specification, [Online]. Available: https://www.trustedcomputinggroup.org/tpm-main-specification/ (visited on 18/07/2016).

[16] A. Nagarajan, V. Varadharajan, M. Hitchens and E. Gallery, 'Property based attestation and trusted computing: analysis and challenges', in *Proceedings of the 2009 Third International Conference on Network and System Security*, ser. NSS '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 278–285, ISBN: 978-0-7695-3838-9. DOI: 10.1109/NSS.2009.83. [Online]. Available: http://dx.doi.org/10.1109/NSS.2009.83.

[17] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy and B. Sniffen, 'Principles of remote attestation', *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[18] F. Armknecht, A.-R. Sadeghi, S. Schulz and C. Wachsmann, 'A security framework for the analysis and design of software attestation', in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: ACM, 2013, pp. 1–12, ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516650. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516650.

[19] Counts, Tracy. (12th May 2014). The OpenAttestation (OAT) Project, [Online]. Available: https://01.org/blogs/tlcounts/2014/openattestation-oat-project (visited on 18/07/2016).

[20] A. Baumann, M. Peinado and G. Hunt, 'Shielding Applications from an Untrusted Cloud with Haven', in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[21] (2011). Drawbridge, [Online]. Available: http://research.microsoft.com/en-us/projects/drawbridge/ (visited on 31/05/2016).

[22] Y. Xu, W. Cui and M. Peinado, 'Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems', in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 640–656. DOI: 10.1109/SP.2015.45.

[23] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz and M. Russinovich, 'VC3: Trustworthy Data Analytics in the Cloud Using SGX', in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 38–54. DOI: 10.1109/SP.2015.10.

[24] V. Costan and S. Devadas, *Intel SGX Explained*, Cryptology ePrint Archive, Report 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.pdf.

[25] R. Paleari, L. Martignoni, G. F. Roglia and D. Bruschi, 'A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators', in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09, Montreal, Canada: USENIX Association, 2009, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855876.1855878.

[26] X. Chen, J. Andersen, Z. M. Mao, M. Bailey and J. Nazario, 'Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware', in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, IEEE, 2008, pp. 177–186.

[27] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich and H. Bos, 'SoK: P2PWNED - Modeling and Evaluating the Resilience of Peer-to-Peer Botnets', *2014 IEEE Symposium on Security and Privacy*, pp. 97–111, 2013, ISSN: 1081-6011. DOI: http://doi.ieeecomputersociety.org/10.1109/SP.2013.17.

[28] D. Andriesse and H. Bos, 'An Analysis of the Zeus Peer-to-Peer Protocol', VU University Amsterdam, Tech. Rep. IR-CS-74, May 2013.

[29] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann and H. Bos, 'Highly resilient peer-to-peer botnets are here: An analysis of Gameover Zeus', *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, pp. 116–123, 2013. DOI: http://doi.ieeecomputersociety.org/10.1109/MALWARE.2013.6703693.

[30] G. Yan, S. Chen and S. Eidenbenz, 'RatBot: Anti-enumeration Peer-to-peer Botnets', in *Proceedings of the 14th International Conference on Information Security*, ser. ISC'11, Xi'an, China: Springer-Verlag, 2011, pp. 135–151, ISBN: 978-3-642-24860-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2051002.2051016.

[31] G. Yan, D. T. Ha and S. Eidenbenz, 'AntBot: Anti-pollution Peer-to-peer Botnets', *Comput. Netw.*, vol. 55, no. 8, pp. 1941–1956, Jun. 2011, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2011.02.006. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2011.02.006.

[32] P. Stewin and I. Bystrov, 'Understanding DMA Malware', in *Detection of Intrusions and Malware, and Vulnerability Assessment: 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, U. Flegel, E. Markatos and W. Robertson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–41, ISBN: 978-3-642-37300-8. DOI: 10.1007/978-3-642-37300-8_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37300-8_2.

[33] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta and I. Koltsidas, 'Implementation and Implications of a Stealth Hard-drive Backdoor', in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13, New Orleans, Louisiana, USA: ACM, 2013, pp. 279–288, ISBN: 978-1-4503-2015-3. DOI: 10.1145/2523649.2523661. [Online]. Available: http://doi.acm.org/10.1145/2523649.2523661.

[34] S. P. Johnson, V. R. Scarlata, C. Rozas, E. Brickell and F. Mckeen. (2016). Intel Software Guard Extensions: EPID Provisioning and Attestation Services, [Online]. Available: https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services (visited on 20/06/2016).

[35] I. Anati, S. Gueron, S. P. Johnson and V. R. Scarlata. (2013). Innovative Technology for CPU Based Attestation and Sealing, [Online]. Available: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing (visited on 20/06/2016).

[36] (2016). Intel Software Guard Extensions, Enclave Writer's Guide, [Online]. Available: https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf (visited on 27/06/2016).

[37] (2016). Intel 64 and IA-32 Architectures Software Developers Manual, Vol. 3D, System Programming Guide, Part 4, [Online]. Available: http://www.intel.nl/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf (visited on 08/07/2016).

[38] (2016). Intel SGX SDK user guide Windows sgx_report_body_t, [Online]. Available: https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Content/sgx_report_body_t.htm (visited on 07/07/2016).

[39] (2016). Intel SGX SDK user guide Windows sgx_quote_t, [Online]. Available: https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Content/sgx_quote_t.htm (visited on 07/07/2016).

[40] (2016). Step by Step Enclave Creation, [Online]. Available: https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Content/Step%20by%20Step%20Enclave%20Creation.htm (visited on 27/06/2016).

[41] (2016). Enclave Definition Language Syntax, [Online]. Available: https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Content/Enclave%20Definition%20Language%20File%20Syntax.htm (visited on 27/06/2016).

[42] C. Kolbitsch, E. Kirda and C. Kruegel, 'The Power of Procrastination: Detection and Mitigation of Execution-stalling Malicious Code', in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, Chicago, Illinois, USA: ACM, 2011, pp. 285–296, ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046740. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046740.

[43] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang and D. Han, 'OpenSGX: An Open Platform for SGX Research', in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2016.

[44] W. Salusky and R. Danford. (13th Jul. 2007). The Honeynet Project: Know Your Enemy: Fast-Flux Service Networks, [Online]. Available: https://www.honeynet.org/papers/ff (visited on 27/06/2016).

[45] (2016). GDB: The GNU Project Debugger, [Online]. Available: https://www.gnu.org/software/gdb/ (visited on 07/07/2016).

[46] (2016). Linux Programmer's Manual CONNECT(2), [Online]. Available: http://man7.org/linux/man-pages/man2/connect.2.html (visited on 07/07/2016).

[47] (2016). Linux Programmer's Manual IP(7), [Online]. Available: http://man7.org/linux/man-pages/man7/ip.7.html (visited on 07/07/2016).

[48] (2016). Intel 64 and IA-32 Architectures Software Developers Manual, Vol. 2B, Instruction Set Reference, N-Z, [Online]. Available: https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf (visited on 20/06/2016).

[49] Intel. (2016). Products (Formerly Skylake), [Online]. Available: http://ark.intel.com/products/codename/37572/Skylake (visited on 18/07/2016).

[50] B. C. Vattikonda, S. Das and H. Shacham, 'Eliminating Fine Grained Timers in Xen', in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11, Chicago, Illinois, USA: ACM, 2011, pp. 41–46, ISBN: 978-1-4503-1004-8. DOI: 10.1145/2046660.2046671. [Online]. Available: http://doi.acm.org/10.1145/2046660.2046671.

[51] Yang, Oliver. (9th Sep. 2015). Pitfalls of TSC usage, [Online]. Available: http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/#tsc-emulation-on-different-hypervisors (visited on 19/07/2016).

[52] (2016). Intel Software Guard Extensions Evaluation SDK, [Online]. Available: https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Default.htm (visited on 27/06/2016).

[53] (2016). Intel Software Guard Extensions for Linux OS, [Online]. Available: https://01.org/intel-softwareguard-extensions (visited on 15/07/2016).

[54] (2016). Intel Software Guard Extensions for Linux OS, linux-sgx, [Online]. Available: https://github.com/01org/linux-sgx (visited on 15/07/2016).

[55] (2016). Intel Software Guard Extensions for Linux OS, linux-sgx-driver, [Online]. Available: https://github.com/01org/linux-sgx-driver (visited on 15/07/2016).

[56] (2016). The Edger8r Tool, [Online]. Available: https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Content/The%20Edger8r%20Tool.htm (visited on 27/06/2016).

[57] Ardillo. (2016). OpenSGX - Bug: Intra Attestation Fail), [Online]. Available: https://github.com/sslab-gatech/opensgx/issues/40 (visited on 18/07/2016).

[58] ——, (2016). OpenSGX - Bug: GDB python scripts indented wrong), [Online]. Available: https://github.com/sslab-gatech/opensgx/issues/45 (visited on 18/07/2016).