Practical Implications of Graphene-SGX

Research Project 2

August 13, 2019

Robin Klusman
University of Amsterdam
Security and Network Engineering
robin.klusman@os3.nl

Derk Barten
University of Amsterdam
Security and Network Engineering
derk.barten@os3.nl

Supervisor:
Gijs Hollestelle
ghollestelle@deloitte.nl

Abstract

Cloud computing is gaining widespread adoption, and with it, the trust that has to be placed in cloud providers is increasing. Intel Software Guard Extensions provide a solution to the issue of having to place this trust by allowing application developers to run their software in a trusted enclave on an untrusted Operating System. Adapting software to SGX in many cases requires significant development effort. Graphene-SGX provides a framework that instead allows running arbitrary unmodified Linux software in SGX enclaves, with the aim to reduce the required development effort. However, such software is often written with the notion that the OS is trusted, and this threat model misalignment with SGX gives rise to possible security issues.

This study explores the implications of running arbitrary applications in Graphene-SGX. We explore both the security implications and the maturity. Furthermore, we aim to provide insight into the issues that exist when running applications in Graphene-SGX. We show that security implications indeed exist; time and date manipulation by an untrusted malicious OS can bypass security measures of the application in certain cases. Moreover, modification of environment variables also presents a possible tampering opportunity for the OS in specific applications. We conclude that Graphene-SGX is currently not at a maturity level that is ready for production environments. Getting arbitrary applications running in Graphene-SGX is not trivial, as there are various bugs and undocumented issues that may arise when attempting to do so.

I. Introduction

In recent years, cloud computing is becoming ever more prevalent. Instead of deploying and maintaining one's own hardware, renting *Infrastructure as a Service* (IaaS) style cloud services, for example, is becoming an increasingly cost-effective alternative, resulting in widespread adoption [5, 25]. Widespread use of such services comes with one important caveat; the application owner no longer has full control over the hardware and *Operating System* (OS) used to run their application. Instead, the cloud provider has full control, and hence, they are responsible for a significant part of the application's security. This requires that one must trust the provider to deliver this security and moreover, trust that they themselves are not malicious. This leads to a trust issue, where the client is forced to place significant trust in the provider.

Intel attempts to solve this trust issue between client and provider with the introduction of Intel *Software Guard Extensions* (SGX). SGX is available in Intel microprocessors starting from the sixth generation Intel Core series microprocessors based on the Skylake microarchitecture which launched in 2015 [11]. With SGX, a client can run their software in a secure container, referred to as an *enclave*, which protects the code being run within from the provider controlled Operating System. An enclave can be regarded as a special memory region that is protected by the CPU and can thus not be accessed with regular memory operations from the OS. SGX introduces a small set of instructions that allow for heavily regulated interaction between the OS and enclave. We will further discuss the architecture of SGX in Section V-A. As a result of SGX, the OS no longer needs to be trusted; one instead places trust in the CPU hardware and SGX implementation of Intel [13].

However, multiple attacks have shown that the protection mechanisms that Intel SGX provides are not foolproof; in several situations, the protections of an SGX enclave can be broken and information can be leaked [6, 9, 15, 26, 31, 38, 46, 54, 55, 56, 58, 60]. These attacks will be further outlined in Section V-C.

Running applications in an SGX enclave usually requires modifications to the application, and for this purpose, Intel provides an SGX *Software Development Kit* (SDK). Alternatively, multiple tools exist that facilitate running applications in SGX enclaves without code modifications, such as Graphene-SGX, SGX-LKL, SCONE, and Panoply [52, 42, 2, 49]. This approach reduces development effort and may lead to quicker and easier adoption of SGX. Graphene-SGX, SGX-LKL, and Panoply work by implementing a Library OS which provides an abstraction layer between the SGX interface and the application that is to be run in the enclave. SCONE, on the other hand, uses Docker to implement secure containers that run in SGX enclaves. Each solution achieves the possibility to run arbitrary software in SGX enclaves, however, at varying rates of performance and security.

The possibility of running unmodified applications in SGX reduces the development effort required to run an application in SGX, and hence, is an appealing option. However, most standard applications are developed with the notion that the Operating System is trusted, while SGX has a threat model that contradicts this notion. Naively running unmodified applications in SGX, with added security and isolation as the objective, may therefore not yield the expected results. It may not be evident to companies or developers considering SGX for their applications, that this is the case.

This study investigates the possible implications that arise when running arbitrary applications in Graphene-SGX. We investigate both the security implications as well as the maturity of the Graphene-SGX framework. Furthermore, we aim to provide new insights for developers opting to use Graphene-SGX in the future.

We chose to investigate Intel SGX, as it is one of the more recent implementations providing software attestation, which ensures the client that it is communicating with the legitimate, trusted execution environment. Additionally, at the time of writing Intel processors have a majority market share in server hardware [14]. The choice for Graphene-SGX was made because it is open source and actively maintained, unlike other tools such as SCONE or Panoply [52, 2, 49].

Our main contributions are as follows:

- We introduce two novel attacks on applications running inside SGX using Graphene-SGX and discuss their mitigations. Furthermore, we show that developers should take special care in implementing applications in Graphene-SGX.
- We evaluate the maturity of Graphene-SGX and give insight to developers on the existing issues that arise when attempting to run applications in Graphene-SGX.
- We present a summary of existing attacks on SGX which should be taken into account by developers.

The remainder of this paper is structured as follows: our research questions are stated in Section II. The ethical implications are then covered in Section III. We next discuss the related work in Section IV, followed by background on Intel SGX and Graphene-SGX in Section V. We then cover the experiments and results for possible security implications of SGX and Graphene-SGX in Section VI. We discuss our findings and give our conclusions in Section VII. We conclude with suggestions for future work in Section VIII.

II. Research questions

Following from the above introduction, we define our research question as follows:

What are the practical implications of running arbitrary applications in Intel SGX using Graphene-SGX?

To answer our main research question we define the following sub-questions:

- What are the workings and limitations of Intel SGX and Graphene-SGX?
- What security implications exist when running applications using Graphene-SGX?
- What is the maturity of Graphene-SGX as a framework for running arbitrary applications?

III. Ethical considerations

Our research is conducted on a dedicated server that is not used by any other users. We do not use any shared hardware, and as such, do not run the risk of interfering with or compromising any third party operations or data.

We believe the impact of our findings will not affect the security of production software utilizing Graphene. The attacks require software that utilizes time or environment variables in a security critical manner, moreover, application level mitigations are possible. Lastly, we believe that the adoption of Graphene for production purposes is minimal.

IV. Related work

Costan and Devadas provide a thorough investigation and detailed explanation of the workings of Intel *Software Guard Extensions* (SGX) [13]. They do so by taking the information available in the literature that introduced the concepts used by Intel SGX [1, 19, 36], and Intel's own documentation [10, 12] and patents [23, 35].

Arnautov et al. developed *Secure CONtainer Environment* (SCONE), a framework for running arbitrary user-space software in Docker containers within an Intel SGX enclave, thereby shielding it from privileged system software [2]. To minimize the *Trusted Computing Base* (TCB), SCONE only exposes a secure interface to the standard C library, `libc`, inside the secure container. This also ensures that the secure containers can run unmodified software inside. To increase performance inside the containers, SCONE implements its own threading implementation as well as asynchronous system calls. Special care is taken in shielding the I/O interfaces of an application inside a secure container from untrusted system software.

Tsai et al. developed a Library OS for use with Intel SGX, called Graphene-SGX which we hereafter refer to as simply Graphene. Graphene allows users to run applications in SGX enclaves without needing to modify the application or incurring crippling overheads on execution [52]. In fact, Graphene shows similar performance to the previously mentioned SCONE but does have a significantly larger sized TCB. This is mainly caused by the decision to use `glibc` and provide a more extensive set of features compared to SCONE. Similar to SCONE, Graphene shields enclave processes from the host OS. However, where SCONE uses a statically linked `libc`, Graphene allows for dynamically linked libraries by verifying the integrity of those dynamic libraries. We discuss the architecture and features of Graphene in more detail in Section V-B.

Several attacks on Intel SGX have already been published, such as various attacks related to caching [6, 9, 15, 26, 31, 38, 46, 56], and other attacks [54, 55, 58, 60]. A more detailed summary of these existing attacks will be given in Section V-C.

V. Background

A. Intel SGX architecture

The Intel *Software Guard Extensions* (SGX) are a set of instructions and mechanisms for memory access that is present in modern Intel processors [10]. The instructions allow for applications to create protected containers, more commonly known as enclaves. An enclave is a region of the memory address space where the processor guarantees integrity and confidentiality. These guarantees are achieved through prohibiting access to enclave memory by the OS or any other application, allowing for secure remote computation on untrusted machines. Such features can be beneficial for customers of cloud providers; further purposes are covered in Section V-D. We use the Intel documentation and the work by Costan and Devadas to discuss the architecture of Intel's SGX and how the integrity and confidentiality guarantees are achieved [10, 13]

Note that we use the term *Operating System* (OS) to refer to the software that runs outside of SGX. This term is used out of convenience; it does not necessarily imply that what is run outside of SGX is always an OS, as is can also be a hypervisor or any other *system software* (i.e. software running in ring 0 or -1).

1) Memory structures: Intel SGX uses a dedicated memory region called the *Processor Reserved Memory* (PRM) which is allocated exclusively for SGX enclaves [13]. The CPU prohibits anything outside the associated enclave from accessing this region. Consequently, the OS and other applications are not allowed access. The PRM contains the *Enclave Page Cache* (EPC), which is a storage structure that contains the enclaves' data.

The EPC storage space is partitioned into memory pages of 4KB. Each EPC page can only be accessed by the enclave that owns it. As EPC pages can only belong to a single enclave, they cannot be used for inter-enclave communication.

Because Intel provides support for multiple enclaves on the same system, EPC page ownership has to be recorded. Partly for this purpose, SGX uses an *Enclave Page Cache Map* (EPCM), wherein the ownership of each EPC page is stated. Additionally, other important information such as the validity of the pages and page permissions are recorded in the EPCM as well [10]. The EPCM can be regarded as a large table where every row represents a single EPC entry.

Enclave metadata is stored in a *SGX Enclave Control Structure* (SECS), which is located on a dedicated EPC page. This data structure contains information related to the enclave's identity, such as the enclave signature.

2) Enclave life cycle: Enclaves have a distinct life cycle that can be divided into four stages: *Creation*, *Loading*, *Initialization*, and *Teardown*.

- An enclave is created when one of the EPC pages is converted into a SECS. The code in an enclave cannot be executed until the `INIT` attribute in the SECS is set to `True`; this attribute is set in the initialization phase.
- In the second phase, the code of the enclave is loaded in EPC pages by the OS. It is only before the enclave is initialized that the OS can access memory of the enclave.
- After the content of the enclave is loaded, the `INIT` attribute of the SECS is set to `True`. Setting this attribute is more complex than it may seem at first glance. The OS must request a special token from the Intel *Launch Enclave* (LE). The LE is a privileged Intel enclave which is part of the SGX implementation. The token received from the LE can be used together with a special instruction to set the `INIT` attribute of the enclave to `True`. Costan and Devadas speculate that the LE does not serve a concrete purpose beside the opportunity for Intel to regulate and license SGX enclaves [13].
- When the `INIT` attribute is set, the memory allocated to the enclave can only be accessed from within the enclave and, therefore, the OS can no longer access the PRM. From this point onward, the enclave is in operation and will run the desired software. When the enclave has performed its function, it is destroyed by setting all EPC pages to invalid in the EPCM. The EPC page containing the SECS is the last page to be invalidated, thereby completely removing the enclave.

3) Software attestation: To allow a client to verify that they are communicating with an expected and trusted piece of software inside an enclave, SGX relies on attestation. Software attestation provides proof to the client that the software in an SGX enclave of the provider is as expected. This proof consists of a piece of *attestation data* which identifies the software through a cryptographic hash, and is cryptographically signed to produce an *attestation signature* [13]. The key used to sign the attestation data is known only to the hardware doing the attestation; this can be verified using a certificate that is provided by the hardware manufacturer, i.e. Intel. Due to the complexity of implementing attestation, SGX makes use of specially privileged enclaves to facilitate this process, such as computing the attestation data and signing it. The key used for the verification of the containers is hard-coded into the SGX microcode [13].

The attestation process also plays an important part in provisioning secrets to an enclave by the remote party. During the attestation process, public keys can be exchanged, with which a secure channel can then be established [21].

4) SGX versions: Since the initial release of SGX in 2015, updates and new versions have been released periodically. In 2016, a new version of SGX was released, SGX version 2. This version adds support for dynamic memory management as its major addition [37].

B. Graphene design

Graphene is a port of the Graphene Library OS to add support for Intel SGX and run unmodified software in secure enclaves [53, 52]. Graphene adds support for multi-process applications to SGX by implementing the `fork` and `execve` system calls. Furthermore, *Inter-Process Communication* (IPC) functionality is added, since memory pages cannot be shared in SGX. Additionally, as part of supporting unmodified applications, Graphene implements support for dynamic shared libraries.

1) Library OS: Graphene uses a Library OS to run arbitrary applications in Intel SGX. A Library OS is a type of Operating System that pushes protection mechanisms to the lowest hardware boundaries [34]. This means that typical Operating System services, such as networking or disk I/O, are implemented in the form of libraries together with a set of policies that enforce access control [33]. An application running in Graphene can call these libraries to get certain OS services, if the policies allow it.

2) Manifests: In order to run an application in Graphene, the user must supply a *manifest* which specifies policies such as the resources that are allowed to be used and what files on disk or network locations should be accessible. The manifest also includes other SGX specific values such as the number of threads and the starting virtual address of the enclave [52]. Additionally, trusted files can be specified in the manifest with a hash digest of the file. When opened, Graphene will verify the integrity of the file and reject it if it does not match the registered hash in the manifest. The manifest itself is signed to protect its integrity and prevent the OS from making malicious modifications [52].

3) Enclave runtime environment: An enclave runtime environment consists of a shielding library `libshield.so`, the application executable and the manifest discussed previously. Running an enclave starts with the *Platform Adaption Layer* (PAL), which in turn calls SGX drivers to initialize the enclave. Next, the enclave goes through the attestation procedure and, when satisfactory, the included shielding library loads the Library OS and standard C library [52]. Once the initialization is finished, dynamic libraries are loaded, and the application is started.

4) Dynamic loading: To support dynamic loading of libraries, Graphene generates a unique signature for any combination of executable and dynamically linked libraries [52]. These signatures are verified when the library is loaded. When the check is unsuccessful, the library will not be loaded.

5) Multi-process support: Support for multi-process applications in SGX enclaves is achieved by running a separate enclave and instance of the Library OS for each process. When a process calls `fork` the new enclave is created and the Library OS, together with the application, is loaded into it. Next, attestation is performed and, if successful, the environment can be loaded into the new enclave. The extra work needed to `fork` a process running in Graphene incurs an overhead of about 10,000x the processing time compared to native execution [52].

Message passing is used between the enclaves to facilitate process forking and other kinds of Inter-Process Communication where needed [52]. Multi-process support is necessary when the goal is to run unmodified applications in Graphene, as it is common for a process to create child processes as part of their standard behavior.

6) System call interfaces: Inside a Graphene enclave, the Library OS exposes the standard C library `libc` to the application. This opens up a possibility for so-called *Iago* attacks, which exploit the fact that system calls were not designed with the possibility of a malicious kernel [8]. The Library OS handles the system call itself whenever possible, or calls the PAL which, according to the paper, implements 36 functions of the host system's *Application Binary Interface* (ABI) [52]. PAL again translates these 36 calls into 28 that will be called out to the untrusted OS and may be vulnerable to Iago attacks [52].

Out of these 36 system calls, 28 effectively go out to the untrusted OS. The host OS behavior for 18 of them can be checked for correctness, 6 cannot harm the enclave even if maliciously manipulated, 2 allow a *Denial of Service* (DoS) attack (`FUTEX_WAIT` and `STREAM_POLL`) while 2 more (`VERB_STAT` and `DIR_READ`) can potentially be used to attack the enclave by the OS [52].

In the Graphene source code, it can be seen that a number of included system calls are directly passed through to the host OS. However, the implementation and security of these system calls are not discussed in their paper. Furthermore, frequently used system calls in Linux are implemented through a *Virtual Dynamic Shared Object* (vDSO) for performance reasons, but there is no documentation on how Graphene treats such system calls either.

7) Known security implications: Graphene marks all enclave pages with `rwx` permissions, which means code pages will be writable during execution; this poses a possible security risk. However, such page permissions are necessary to support arbitrary Linux *Executable and Linkable Format* (ELF) executables, which often need writable code pages during the linking phase. While in SGX, page permissions cannot be altered after the creation of an enclave. SGX version 2 solves this problem by supporting dynamic changes in page permissions [52].

C. Security history overview

In recent years, there has been a significant amount of research conducted on Intel SGX. In order to obtain a clear understanding of the current developments and the research leading up to this point, we provide a timeline of the most important publications regarding the security of SGX.

1) September 2013: The programming reference manual for Intel SGX was published on September 26th, 2013 [18, 10].

2) May 2015: In May of 2015, Xu et al. published a *controlled channel* attack on SGX enclaves [60]. A controlled channel attack is a type of side-channel attack, where the adversary has control over the platform and hence can mount more powerful attacks. Their attack relies on the generation of page faults to infer the state of the enclave. It requires prior analysis of the application running in the enclave to identify the page fault patterns. These patterns will be used to infer what code is being executed or what variables are being modified [60].

3) July 2015: The sixth generation of Intel's processor architecture, called Skylake, was released in July 2015, which includes support for native SGX [11].

4) May 2016: Weichbrodt et al. investigate ways to exploit synchronization bugs in applications that run inside enclaves [58]. They noticed a trend to use SGX as a one-size-fits-all isolation and security solution for complete and off the shelf applications. However, these applications are often written with an entirely different threat model from that of SGX. SGX assumes the OS may be malicious, whereas applications often consider the OS as trusted. To demonstrate the exploitability of such synchronization bugs in SGX, they developed a tool called AsyncShock.

AsyncShock uses `mprotect` to instantaneously change the permissions of memory pages in the page table which is used by the *Memory Management Unit* (MMU). This causes a segmentation fault and allows interrupting execution of the enclave with high precision. A custom handler for segmentation faults catches this interrupt and gives control to the attacker when the fault is triggered, thus giving the attacker a more reliable way to exploit synchronization bugs if they exist in the target application.

5) February 2017: Shih et al. introduce mitigations for the controlled channel attack by Xu et al. [48, 60]. They use the *Transactional Synchronization Extensions* (TSX), which allow page faults to be effectively hidden from the OS. This works by the nature of TSX to abort an operation when a fault occurs, while suppressing the notification of the actual fault to the OS. Because the controlled channel attack by Xu et al. relies completely on triggering page faults, this attack is completely mitigated by hiding page faults.

6) August 2017: A side-channel attack was published that uses a technique called *branch shadowing* [31]. This attack works by analyzing the source code of the application, after which the application is run inside an enclave and the *Branch Target Buffer* (BTB) is used to reveal what branches are taken by the application. This allows the attacker to infer the fine-grained control flow of applications within SGX enclaves. With this attack, they were able to disclose RSA keys handled inside an enclave.

In the same month, another two papers were published regarding side-channel attack using CPU caches. Moghimi et al. developed a tool called CacheZoom that can track memory accesses of the SGX enclave through a prime and probe approach targeting the L1 cache [38]. The study demonstrates the effectiveness of their approach by performing an AES recovery attack. Furthermore, the study claims that CacheZoom requires fewer measurements than previous cache side-channel attacks.

Brasser et al. developed a set of novel techniques to perform cache side-channel attacks. They improve on the present prime and probe methods by introducing techniques to reduce noise, which is usually one of the critical drawbacks of prime and probe style attacks. The study also shows that this attack differs from previous cache attacks due to its straightforward deployment and avoidance of conventional detection methods [6].

Van Bulck et al. extend on the work of Xu et al. in [60] by developing a method to track page accesses without triggering page faults [55]. They monitored A(ccessed) and D(irty) flags on memory pages and measured cache timings to identify whether a page has been accessed.

7) October 2017: Jang et al. released a *Denial of Service* (DoS) *Rowhammer* attack that targets Intel SGX enclaves [22]. The study developed a method called *SGX-Bomb*; which is a piece of software that is run inside an SGX enclave. It performs a Rowhammer attack in order to introduce bit-flips in the enclave memory. According to the study, when the CPU detects bit-flips, the whole system will halt. This, therefore, provides a serious DoS opportunity that could be aimed at cloud providers that offer SGX enclaves as a service.

In the same month as the Rowhammer attack, a framework called *SGX-Step* was released [54]. SGX-step is an open-source Linux kernel framework that allows host applications to perform APIC timer interrupts and track page table entries from user-space. According to the study, this framework enables several new or improved attacks.

8) January 2018: Van Bulck et al. developed the Foreshadow transient execution attack which uses an out-of-order execution bug to perform unauthorized memory access [56]. This attack is functional with arbitrary enclave code and does not necessarily require root level access. The attack is related to the Meltdown attack [32]. It allows for enclave memory to leak to the CPU cache, which can later be obtained using the same methods as the Meltdown attack. Foreshadow proved that the security guarantees of Intel's SGX could be broken. Later that year, after further investigation, another two Foreshadow related attacks were published [59].

9) February 2018: Shortly after the disclosure of the Spectre and Meltdown attacks on Intel CPUs in early 2018, Chen et al. published their attack on SGX, called SgxPectre, that uses the same principles [9, 29, 32]. Just like the Spectre vulnerability, SgxPectre uses an exploit leveraging the speculative execution employed by Intel CPUs.

To execute this attack, first, the *Branch Target Buffer* (BTB) needs to be poisoned to set up branch prediction to a location where malicious code is stored. This code is crafted such that it will leak the in-memory secrets through state changes in the cache. Next, the buffers are flushed to enforce branch prediction look-up in the poisoned BTB and delay the detection of the invalid branch prediction. Finally, the malicious instructions are run and, afterwards, the leaked data can be retrieved by observing the state changes.

10) March 2018: Evtyushkin et al. show that branch prediction in Intel CPUs can be used to leak information from other processes without poisoning the Branch Target Buffer [15]. Their attack, called *BranchScope*, is a *prime and probe* style attack. It exploits the fact that the *Branch Prediction Unit* (BPU) is shared between processes executing on the same core and that these co-located processes can leak data between them through the BPU.

To execute the attack, the *Pattern History Table* (PHT) entry of interest first has to be primed by executing some set of branch instructions to get it into a known state. Then, the victim process should be allowed to execute while waiting for the PHT state to change. Then changes in the PHT should be observed by probing while also continuing to prime. These steps allow an attacker to identify what branch direction the victim process has taken for specific branches.

In the same month, Kim et al. publish their extension of the work done in [60] and [55] by introducing the *Version IDentification* (VID) attack and mitigation [26]. Their attack is aimed at disclosing what software is running inside an enclave, even if the binary is encrypted. They show this is feasible by comparing page access patterns which include information about consecutive page accesses in the same page to a set of known patterns.

11) May 2019: Schwarz et al. show that malware running in an SGX enclave can be used to successfully attack another enclave running the `mbedTLS` RSA implementation, and leak the private key. They do so by employing a prime and probe attack, which relies on a cache eviction set to do the priming, and a high-resolution timer to distinguish cache hits from misses for the probing. Although this is not a new attack, their implementation hides the attack code in an enclave and can target other enclaves.

D. Use cases

Intel SGX has been used for a wide range of applications. The confidentiality, integrity and isolation from the OS provided by SGX has attracted a large number of applications; these include a database [43], a key-value store [28], secure system logs [24], Map-Reduce data processing [45, 40], TLS termination [3] and various middle-boxes [17, 51, 16].

Another proposed use-case for SGX is video game protection mechanisms, both anti-cheat solutions and *Digital Rights Management* (DRM) [4]. Similarly, the compilation of closed-source software can be done inside enclaves to protect intellectual property from theft [52]. The protections offered by SGX are particularly useful here, as the execution takes place on the machine of an untrusted user.

Furthermore, there are existing applications that are adapted to run with Intel SGX, such as Tensorflow, Tor and Zookeeper [30, 27, 7].

VI. Practical implications

In this section, we cover the security issues and implementation problems we found when running arbitrary applications in SGX enclaves with Graphene. We first discuss our experimental setup, followed by the security implications, and finally, we cover the maturity of the framework.

A. Experimental setup

Our experiments were conducted on a dedicated server, housing an Intel Core i7-7700 CPU that supports SGX version 1. The Operating System used for our experiments is Ubuntu 16.04. We used the latest release of Graphene, which was version 0.6 at the time of writing [39], which we used solely in debug mode, as we did not have access to a key needed to run in release mode. The Intel SGX Platform Software, SDK and Driver version used is the most current version from the *intel-sgx* Github page at time of writing. This version is equivalent to release version 2.6 which is about to be released in the upcoming days [20].

B. Date / time manipulation

In Section V-B, we discussed Iago attacks on Graphene and the mitigations that Tsai et al. have implemented. They state that only the `FILE_STAT` and `DIR_READ` system calls are exploitable and leave the implementation of mitigations as future work. Moreover, they state that other system calls can, in the worst case, be used for a DoS attack on the enclave [52].

We investigated the `gettimeofday` system call, which is not considered harmful by the authors of Graphene. This system call is handled through an Operating System controlled *Virtual Dynamic Shared Object* (vDSO) [57], as can be seen in the Graphene source code in `Pal/src/host/Linux/db_misc.c`. A vDSO is a small shared library that is automatically mapped in the memory space of every application by the kernel. This feature was introduced to reduce the performance overhead of context-switches caused by frequent system calls from user-space code [57]. We believe modifications to the time and date in the host OS will therefore also be reflected in an enclave when `gettimeofday` is called. This presents an opportunity for a malicious OS to influence the code running in an enclave.

We were able to confirm that time changes in the OS influence the result of the `gettimeofday` system call. We used a simple proof of concept C program running in Graphene that displays the current time to do so. In order to show a more practical scenario, we ran an Apache web server in SGX using Graphene. The server serves a minimal web page where rate limiting is applied through a simple PHP script. This script compares the current time to the time of the previous successful request. Such an approach simulates, for instance, rate limiting on user login attempts in a more complex web application.

To show how time manipulation can be abused to avoid rate limiting, we first request our example webpage multiple times within the same minute. This triggers the rate limiting, which results in a temporary lockout. Next, the server time is bumped ahead by the lockout time, in this case, one minute. Lastly, the webpage is requested again, and this time, the server returns the requested web page. This positive response indicates the attack was successful.

Although the consequences of this example may seem limited, for applications that rely on time as a security critical feature, this attack has a significant impact. It can, for instance, allow expired keys or tokens to be reused in applications that use such a scheme. We discuss the potential impact of these types of attacks below, with Kerberos as an example of a real-world application.

C. Kerberos

The Kerberos authentication protocol works based on timestamped tickets to allow access to resources on a network [50]. We first describe the Kerberos architecture and, afterwards, illustrate how this architecture may be vulnerable when components are run in Graphene.

When a user logs into a Kerberos network, they first contact the *Authentication Service* (AS); this is a service that is provided by the *Key Distribution Center* (KDC). The AS, as the name suggests, authenticates the user. The KDC however, can be regarded as the heart of Kerberos; it contains the symmetric keys of all users and offers the services for authentication. Apart from the AS, the KDC also provides the *Ticket Granting Service*

(TGS), which issues the tickets that are used to contact resources throughout the Kerberos enabled network. When the user has authenticated with the AS, the TGS provides the user with a so-called *Ticket Granting Ticket* (TGT).

This TGT is a special ticket that enables the user to request other tickets from the TGS when needed. These other tickets, called session tickets, are used to access resources throughout the Kerberos enabled network. When a user wants to use a resource on the network, it sends its TGT to the TGS. The TGS then verifies the TGT and, if valid, returns a session ticket and key that can be used to contact the desired resource.

Each TGT has an expiration date and time embedded in the ticket that is evaluated when the ticket's validity is checked by the TGS. The TGS uses its own view of time for this check, and hence if the date or time of the TGS is changed, it may incorrectly validate expired tickets. As such, running the TGS in Graphene on an untrusted Operating System would possibly expose it to exploits involving the reuse of expired tickets. The consequence of incorrectly accepting expired tickets could be significant. It could, for instance, allow ex-employees to access resources on the network after they have left the company.

Apart from the risk of running TGS in SGX using Graphene, we would argue that regular Kerberos resources are vulnerable to the same attack. To avoid replay attacks of session tickets, an additional component, called *authenticators*, are sent along when authenticating to a Kerberos resource. An authenticator contains a timestamp that is encrypted with the session key between the user and the desired resource on the network. The resource can decrypt the authenticator to obtain the timestamp, which is then compared to the current time to assess whether the use of the ticket is valid. Running a resource in SGX will therefore also introduce the same exploit and allows the reuse of expired session tickets, which potentially grants unauthorized access.

D. Environment variable passing

Apart from the use of time-related system calls, the OS can employ additional methods to influence execution in Graphene enclaves. We decided to investigate the effect of environment variables on Graphene applications and how an attacker could abuse this.

Many applications use environment variables to set certain execution parameters. The value and presence of environment variables can, therefore, influence the execution of applications. This is the case, for instance, with variables such as `LD_LIBRARY_PATH` or `LD_PRELOAD` for GCC. Malicious changes to these variables can cause harmful code to be included instead of the intended libraries, which can have consequences for the security of the application.

In Linux, environment variables are by default inherited from the parent process by the child process [47]. Therefore, if Graphene is started with Bash as its parent process, the environment variables defined in Bash are inherited by Graphene. However, an enclave developer can explicitly specify the value of environment variables in the manifest file for their application. If an environment variable is indeed specified in the manifest, its value takes precedence, and the parent process' value, if present, is disregarded. This creates the opportunity for a parent process, such as Bash, to call Graphene with malicious environment variables while hoping they are omitted from the manifest.

TABLE I
THE EFFECT OF ENVIRONMENT VARIABLES SET IN BASH AND MANIFEST
ON THE OUTPUT OF THE GETENV SYSTEM CALL FOR APPLICATIONS RUNNING IN GRAPHENE

No.	Bash	Manifest		getenv
1	`MYENV="foo"`	`MYENV=""`	→	`""`
2	`MYENV="foo"`	N/A	→	`"foo"`
3	N/A	`MYENV=""`	→	`""`
4	N/A	N/A	→	`NULL`

To test the behavior of environment variables in Graphene, we created the proof of concept C program shown in Appendix A, which prints the content of the `MYENV` environment variable. The program is rather uncomplicated; it calls the `getenv` system call and prints the result to `stdout`. We evaluate the four combinations of setting and not setting the `MYENV` environment variable, both in the manifest file and in Bash. The results of our experiments are summarized in Table I above.

- In the first experiment, we set the environment variable `MYENV` in Bash to the value `foo`; while in the manifest file, it is set to the empty string. When `getenv` is called for `MYENV`, an empty string is returned.
- The second experiment is similar to the first experiment, with the exception that `MYENV` is not set in the manifest file. We observe that `getenv` returns the value `foo`, which is inherited from the Bash parent process.
- The third experiment removes the definition of `MYENV` in Bash and sets `MYENV` to an empty string in the manifest. As expected, we observe that that `getenv` returns an empty string.
- In the last experiment, we neither set `MYENV` in the manifest file nor in Bash. Here, `getenv` returns the value `NULL`, which is per the specification of `getenv` when no environment variable is set.

Our experiments confirm that environment variables not explicitly set to be empty in the manifest file, are carried over into the application running in Graphene unfiltered.

Apart from the proof of concept, we have performed this attack in practice on a real-world application by using one of the included applications of Graphene, namely the *GNU Compiler Collection* (GCC). Although, as the name suggests, GCC contains a collection of compilers, we will be limiting ourselves to the C compiler. GCC contains several environment variables that influence its behavior during compilation. We have chosen to use the `SOURCE_DATE_EPOCH` environment variable as an example to influence the recorded compilation date and time of the output binary [41]. This environment variable could, for instance, be abused to set a date in the future, thereby possibly triggering automatic updates if the software includes an auto-update feature based on the compilation date of the binary.

Modifications to `SOURCE_DATE_EPOCH` will update the GCC preprocessor definition of the `__TIME__` and `__DATE__` macro in C source code [41]. Macros are commonly used in software development and provide an additional layer of indirection that the programmer can use to their advantage. The macro definitions are resolved during the preprocessing stage to conform to the C syntax. We have created a small C program that prints the content of the `__DATE__` macro to standard output. When GCC is compiling with the `SOURCE_DATE_EPOCH` variable set to the value zero, we can see that the content of the `__DATE__` definition is resolved to the same value. In our C program, the string outputs zero as well. Note that zero corresponds to Unix timestamp zero, which corresponds to the first second of 1970. Without the `SOURCE_DATE_EPOCH` set, the program outputs the current Unix timestamp, which corresponds to the first of July 2019 at the time of writing.

Even though our example used `SOURCE_DATE_EPOCH`, one can imagine that the potential impact of other environment variables, such as `LD_LIBRARY_PATH`, `LD_PRELOAD` or `LD_DEBUG`, is more severe. However, we found that common environment variables, including `LD_LIBRARY_PATH`, are present in example manifest files provided by Graphene. Therefore, developers are more likely to include these variables in their manifest files by default as well. Thus, using these common variables maliciously may be difficult.

E. Framework maturity

In this section, we discuss the practical implications of Graphene related to the maturity of the framework. We investigate whether the current state of Graphene is sufficient for use in production environments. We base this judgement on the implemented system calls, stability, and ease of use. These are catagories that we believe are useful to provide a clear picture of Graphene's maturity and its suitability for production use. However, it must be noted that this is not a universal method to specify maturity of software but it suits our needs.

1) System calls: Because Graphene acts as a layer between the application inside SGX and the operating system, it requires the implementation of a large set of system calls. The system calls are implemented in two different ways; a total of 145 system calls are implemented directly in the Library OS, and are defined in `shim_syscalls.c`. Another 165 system calls are passed through to the operating system directly. We verified that there is no overlap in system calls between the two categories; this results in a total of 310 implemented system calls.

In Ubuntu 16.04 with Linux kernel `4.18.0-22`, a total of 326 system calls are defined in `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`. All system calls defined as pass-through in Graphene, are also present in Linux. In contrast, a few of the Library OS implemented system calls are not present in Linux. This absence of a few system calls can be explained by the fact that Library OS contains several internal, Graphene specific system calls that are not present in Linux.

Additionally, 24 Linux system calls are neither present as pass-through nor implemented in the Library OS, but are defined in Linux. Of these system calls, six system calls are designated as unimplemented in Linux. This results in a total of 18 system calls that are unaccounted for in Graphene, which we list below.

- `bpf`
- `epoll_ctl_old`
- `epoll_wait_old`
- `execveat`
- `finit_module`
- `getrandom`
- `kcmp`
- `kexec_file_load`
- `membarrier`
- `memfd_create`
- `mlock2`
- `process_vm_readv`

- `process_vm_writev`
- `renameat2`
- `sched_getattr`
- `sched_setattr`
- `seccomp`
- `userfaultfd`

We have not encountered issues with these missing system calls with the applications that we have tested. We, therefore, expect that these system calls are not as widely used, which could be the reason for their omission from Graphene.

Because most of the 326 Linux system calls are implemented, this gives the impression that there should be little system call related issues when running applications in Graphene. However, we found that not all system calls work as intended. When we deployed SQLite3 in Graphene, for instance, opening an existing database on disk is prohibited by the Library OS, while said database file is correctly specified in the manifest.

2) Stability: Stability is an important aspect when running software in production environments, as bugs or downtime can be costly. When running applications in Graphene such as Python, SQLite3, and Nginx, we encountered some critical errors that prevented the application from continuing standard operation.

When running a Python Flask web application, the use of `sys.urandom` in this package resulted in the error `[Errno 14] Bad address`. Writing our own implementation of the `urandom` function did not fix the issue and instead yielded an assertion error in Graphene.

As stated before, when running SQLite3, we were unable to open a database file from disk. Additionally, Nginx had the issue that it would freeze after the first time a user loads a page, requiring a restart of the enclave before a page could be requested again.

3) Ease of use: Covering the ease of use of software is difficult as it tends to rely on subjective experience. We cover what applications were successfully run in Graphene during our research and in what environment this was run. It may very well be that the success heavily depends on different kernel versions, as Graphene relies on a broad range of system calls.

We have deployed Graphene in both Ubuntu version 18.04 and 16.04. The documentation of Graphene recommends Ubuntu 14.04 or 16.04 along with specific kernel versions 3.5, 3.14 or 4.4, as these should be functioning correctly [39]. Despite this recommendation, we first attempted to run Graphene on the latest long term Ubuntu release, as the best practice of using the most up to date version of software applies to Operating Systems as well.

With Ubuntu 18.04 and Linux kernel version `4.18.0-22`, we encountered many issues with Graphene. Attempting to run the Graphene supplied test applications such as Apache, Python, Nginx, and Lighttpd resulted in only Apache being able to run successfully. For Nginx and Lighttpd, errors occur already during compilation or right away during execution. In the case of Python, it seems to depend on what Python modules are used if an error occurs or not.

Apart from the supplied test applications, we attempted to run MySQL in Graphene on Ubuntu 18.04 but were not able to get this working. However, we were able to run SQLite3 successfully, although only without using an existing database.

In 16.04 with kernel 4.15.0, we were able to run Apache, MySQL, Nginx, and Lighttpd without crashing immediately. We encountered significantly less internal errors in using Ubuntu 16.04 as opposed to Ubuntu 18.04. However, Nginx, MySQL, and Lighttpd still showed issues that kept us from running them in a stable manner. We were able to run Apache without running into issues with our test web pages.

Another notable aspect of Graphene is that all resources such as CPU threads or files on disk that are read or written to during execution, need to be known a priori and specified correctly in the manifest file. Without this, the application will likely not be able to run correctly. For this reason, developers that attempt to run applications in Graphene need have extensive knowledge of their application's needs during execution. Graphene does not alleviate the amount of effort or level of know-how needed to implement applications in SGX correctly.

VII. Discussion

In the previous section, we have shown that there are multiple implications when running arbitrary applications in Graphene. Using practical examples, we showed that clock manipulation and maliciously setting environment variables by an adversarial OS has the potential to compromise the security of applications running inside an SGX enclave using Graphene. Mitigations do exist for these attacks, clock manipulation attacks by the OS can be neutralized by relying on a trusted (remote) service to supply time over a secure channel. Similarly, malicious environment variables can be mitigated by explicitly setting all relevant variables in the manifest, thereby not leaving any opportunities for manipulation by the OS.

Intel SGX does not claim to protect against any variant of side-channel attacks, which Iago attacks such as clock manipulation or malicious environment variables can be characterized as; hence, we do not fault SGX for being susceptible to such attacks. Graphene, on the other hand, does claim that they have mitigated the threat of Iago attacks through system calls by implementing checks on host behavior, except for `FILE_STAT` and `DIR_READ`. As we have shown, there are opportunities to exploit other system calls, but they are heavily application dependent and can be mitigated.

However, the necessity for such mitigations is not apparent; we expect developers may not be fully aware of the potential vulnerabilities that still exist when running their software in Graphene and SGX. Hence, we do not fault Graphene for the absence of mitigations, but we do believe Graphene should better inform developers of the possibility for attacks. Additionally, SGX has a rich vulnerability history. Although these vulnerabilities are patched with incremental security updates, this history may be prove to be a good indication for the future. If so, developers should be cautious with the use of Intel SGX for highly sensitive applications on untrusted hardware. For such applications, the use of exclusively owned hardware may be preferable.

Based on our preliminary results with a limited set of applications, we believe that Graphene is not at a maturity level that is ready for production purposes. We did not perform exhaustive testing on the broad range of functionality provided by Graphene, but with the applications we tested, significant effort was required to get the applications running. In other cases, applications were not able to run correctly at all due to issues in the internals of Graphene.

Graphene was developed mainly as a research tool, and hence, we cannot expect the level of stability we would expect from a commercial piece of software. However, this is not obvious from the documentation, and therefore, developers may have different expectations when first opting to use Graphene. Another important thing to note is that our experiments were conducted while running SGX in debug mode, as we did not have an Intel approved key needed to run in release mode. To the best of our knowledge, running in debug mode has no significant influence on the results of our performed experiments. However, we cannot decisively state this, as we have not done comparative testing in release mode.

In this study, we did not look at the performance implications of running applications in Graphene and SGX. This aspect has already been extensively covered in the original Graphene-SGX paper [52], and hence we refer to that paper for performance related-implications.

A. Conclusion

As becomes clear from our findings, there are significant implications when naively running arbitrary software in Graphene, both in terms of security and usability. A malicious Operating System can compromise enclave security in specific scenarios if no mitigations are implemented in the application itself. Furthermore, running an application in Graphene requires significant effort and know-how by the user. We deem Graphene to not be at a maturity level where it can be used for production purposes, as still numerous bugs and other issues exist. Graphene in our opinion does not provide a secure and stable framework to run applications. We do, however, believe that it provides a decent open-source tool to conduct further research on SGX. We, therefore, advise any developers opting to run their applications in Graphene to take particular care when doing so, while also taking note of the security history of SGX.

VIII. Future work

This study showed how the reliance on the OS for the current time could prove to have security consequences in specific applications. A similar issue may be present with applications requiring randomness, for which they usually rely on the OS. For instance, OpenSSL uses randomness from `/dev/urandom` to create an initial seed for random number generation on UNIX-like operating systems [44]. Because the OS controls the random number generation, it could compromise the randomness and thereby cause weak key generation. Therefore, we think that investigation into in random number generation in Graphene, and the effect on applications running in Graphene would be valuable.

In this study, we did not investigate other tools that allow unmodified applications to run in SGX enclaves, such as SCONE and Panoply [30, 49]. Exploring such tools and comparing their security and maturity to that of Graphene would be valuable, as developers can then choose the tool most suitable to their needs. Additionally, insight into how other tools implement certain functionality or security measures may prove useful for the field as a whole, as it can be used to improve all similar tools.

Acronyms

ABI	Application Binary Interface. 6

AS	Authentication Service. 11
BPU	Branch Prediction Unit. 9
BTB	Branch Target Buffer. 8, 9
DoS	Denial of Service. 6, 8, 10
DRM	Digital Rights Management. 9
ELF	Executable and Linkable Format. 7
EPC	Enclave Page Cache. 4, 5
EPCM	Enclave Page Cache Map. 4, 5
GCC	GNU Compiler Collection. 12, 13
GNU	GNU's Not Unix. 13, 17
IaaS	Infrastructure as a Service. 1
IPC	Inter-Process Communication. 5, 6
KDC	Key Distribution Center. 11
LE	Launch Enclave. 4
MMU	Memory Management Unit. 7
OS	Operating System. 1, 3–7, 9–12, 14–17
PAL	Platform Adaption Layer. 6
PHT	Pattern History Table. 9
PRM	Processor Reserved Memory. 4, 5
SDK	Software Development Kit. 1
SECS	SGX Enclave Control Structure. 4, 5
SGX	Software Guard Extensions. 1–12, 14–18
TCB	Trusted Computing Base. 3
TGS	Ticket Granting Service. 11
TGT	Ticket Granting Ticket. 11
TSX	Transactional Synchronization Extensions. 7
vDSO	Virtual Dynamic Shared Object. 6, 10
VID	Version IDentification. 9

References

[1] Ittai Anati et al. "Innovative technology for CPU based attestation and sealing". In: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. ACM New York, NY, USA. 2013.

[2] Sergei Arnautov et al. "{SCONE}: Secure Linux Containers with Intel {SGX}". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 689–703.

[3] Pierre-Louis Aublin et al. "TaLoS: Secure and transparent TLS termination inside SGX enclaves". In: *Imperial College London, Tech. Rep* 5 (2017).

[4] Erick Bauman and Zhiqiang Lin. "A case for protecting computer games with sgx". In: *Proceedings of the 1st Workshop on System Software for Trusted Execution*. ACM. 2016, p. 4.

[5] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. “Cloud computing: A study of infrastructure as a service (IAAS)”. In: *International Journal of engineering and information Technology* 2.1 (2010), pp. 60–63.

[6] Ferdinand Brasser et al. “Software Grand Exposure:{SGX} Cache Attacks Are Practical”. In: *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*. 2017.

[7] Stefan Brenner et al. “Securekeeper: confidential zookeeper using intel sgx”. In: *Proceedings of the 17th International Middleware Conference*. ACM. 2016, p. 14.

[8] Stephen Checkoway and Hovav Shacham. “Iago attacks: Why the system call api is a bad untrusted rpc interface”. In: *ASPLOS*. Vol. 13. 2013, pp. 253–264.

[9] Guoxing Chen et al. “Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution”. In: *arXiv preprint arXiv:1802.09085* (2018).

[10] Intel Corperation. *Software guard extensions programming reference, revision 2*. 2014.

[11] Intel Corperation. *6th Generation Intel Processor Family Specification Update*. 2018.

[12] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. 2015.

[13] Victor Costan and Srinivas Devadas. “Intel SGX Explained.” In: *IACR Cryptology ePrint Archive* 2016.086 (2016), pp. 1–118.

[14] DIGITIMES. *AMD to challenge Intel server processor market dominance*. URL: www.digitimes.com/news/a20190328PD200.html (visited on 06/01/2019).

[15] Dmitry Evtyushkin et al. “Branchscope: A new side-channel attack on directional branch predictor”. In: *ACM SIGPLAN Notices*. Vol. 53. 2. ACM. 2018, pp. 693–707.

[16] David Goltzsche et al. “Endbox: scalable middlebox functions using client-side trusted execution”. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2018, pp. 386–397.

[17] Juhyeng Han et al. “Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module”. In: *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM. 2017, pp. 99–105.

[18] Matthew Hoekstra. *Intel SGX for Dummies*. URL: https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx (visited on 06/07/2019).

[19] Matthew Hoekstra et al. “Using innovative instructions to create trustworthy software solutions.” In: *HASP@ ISCA* 11 (2013).

[20] intel. *GitHub - intel/linux-sgx: Intel SGX for Linux**. URL: https://github.com/intel/linux-sgx (visited on 06/26/2019).

[21] *Intel SGX Developer Guide*. URL: https://software.intel.com/sites/default/files/managed/33/70/intel-sgx-developer-guide.pdf (visited on 08/12/2019).

[22] Yeongjin Jang et al. “SGX-Bomb: Locking down the processor via Rowhammer attack”. In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM. 2017, p. 5.

[23] Simon P Johnson et al. *Technique for supporting multiple secure enclaves*. US Patent 8,972,746. Mar. 2015.

[24] Vishal Karande et al. "Sgx-log: Securing system logs with sgx". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM. 2017, pp. 19–30.

[25] Ali Khajeh-Hosseini, David Greenwood, and Ian Sommerville. "Cloud migration: A case study of migrating an enterprise it system to iaas". In: *2010 IEEE 3rd International Conference on cloud computing*. IEEE. 2010, pp. 450–457.

[26] Deokjin Kim et al. "SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure". In: *Computers & Security* 82 (2019), pp. 118–139.

[27] Seongmin Kim et al. "Enhancing security and privacy of tor's ecosystem by using trusted execution environments". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 145–161.

[28] Taehoon Kim et al. "ShieldStore: Shielded In-memory Key-value Storage with SGX". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM. 2019, p. 14.

[29] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[30] Roland Kunkel et al. "TensorSCONE: A Secure TensorFlow Framework using Intel SGX". In: *arXiv preprint arXiv:1902.04413* (2019).

[31] Sangho Lee et al. "Inferring Fine-grained Control Flow Inside {SGX} Enclaves with Branch Shadowing". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 557–574.

[32] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[33] Anil Madhavapeddy and David J Scott. "Unikernels: Rise of the virtual library operating system". In: *Queue* 11.11 (2013), p. 30.

[34] Anil Madhavapeddy and David J. Scott. *Unikernels: Rise of the Virtual Library Operating System*. URL: https://queue.acm.org/detail.cfm?id=2566628 (visited on 06/26/2019).

[35] Francis X McKeen et al. *Method and apparatus to provide secure application execution*. US Patent 9,087,200. July 2015.

[36] Frank McKeen et al. "Innovative instructions and software model for isolated execution." In: *Hasp@ isca* 10.1 (2013).

[37] Frank McKeen et al. "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*. ACM. 2016, p. 10.

[38] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "Cachezoom: How SGX amplifies the power of cache attacks". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 69–90.

[39] oscarlab. *GitHub - oscarlab/graphene: Graphene / Graphene-SGX - a library OS for Linux multi-process applications, with Intel SGX support*. URL: https://github.com/oscarlab/graphene (visited on 07/05/2019).

[40] Rafael Pires et al. "A lightweight MapReduce framework for secure processing with SGX". In: *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press. 2017, pp. 1100–1107.

[41] GNU Press. *GCC 9.1 Manual*. 2019.

[42] Christian Priebe and Peter Pietzuch. *Software Guard Extensions Linux Kernel Library (SGX-LKL)*. URL: https://github.com/lsds/sgx-lkl (visited on 06/01/2019).
[43] Christian Priebe, Kapil Vaswani, and Manuel Costa. "Enclavedb: A secure database using sgx". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 264–278.
[44] *Random Numbers - OpenSSLWiki*. URL: https://wiki.openssl.org/index.php/Random_Numbers (visited on 07/09/2019).
[45] Felix Schuster et al. "VC3: Trustworthy data analytics in the cloud using SGX". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 38–54.
[46] Michael Schwarz et al. "Malware guard extension: Using SGX to conceal cache attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2017, pp. 3–24.
[47] *Shell Execution Environment*. URL: http://pubs.opengroup.org/onlinepubs/7908799/xcu/chap2.html#tag_001_012 (visited on 07/06/2019).
[48] Ming-Wei Shih et al. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." In: *NDSS*. 2017.
[49] Shweta Shinde et al. "Panoply: Low-TCB Linux Applications With SGX Enclaves." In: *NDSS*. 2017.
[50] Jennifer G Steiner, B Clifford Neuman, and Jeffrey I Schiller. "Kerberos: An Authentication Service for Open Network Systems." In: *Usenix Winter*. Citeseer. 1988, pp. 191–202.
[51] Bohdan Trach et al. "ShieldBox: Secure middleboxes using shielded execution". In: *Proceedings of the Symposium on SDN Research*. ACM. 2018, p. 2.
[52] Chia-Che Tsai, Donald E Porter, and Mona Vij. "Graphene-SGX: A Practical Library {OS} for Unmodified Applications on {SGX}". In: *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17)*. 2017, pp. 645–658.
[53] Chia-Che Tsai et al. "Cooperation and security isolation of library OSes for multi-process applications". In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 9.
[54] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A practical attack framework for precise enclave execution control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM. 2017, p. 4.
[55] Jo Van Bulck et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 1041–1056.
[56] Jo Van Bulck et al. "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 991–1008.
[57] *VDSO(7) Linux Programmer Manual*. 2018.
[58] Nico Weichbrodt et al. "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves". In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 440–457.
[59] Ofir Weisse et al. *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*. Tech. rep. Technical report, 2018.

[60] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 640–656.

APPENDIX A
ENVIRONMENT VARIABLE C SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    char* s = getenv("MYENV");
    if (s != NULL)
        printf("%s\n", s);
    else
        printf("MYENV does not exist\n");
    return 0;
}
```