Institut für Betriebssysteme und Rechnerverbund

Technische Universität Braunschweig

Tools and System Support for Advanced Enclave Programming

Nico Weichbrodt

February 19, 2024

Tools and System Support for Advanced Enclave Programming

Der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)

vorgelegte Dissertation
(kumulative Arbeit)

von
Nico Weichbrodt
geboren am 26. Januar 1991
in Wolfenbüttel

Eingereicht am:	19. Februar 2024
1. Referent:	Prof. Dr.-Ing. Rüdiger Kapitza
2. Referent:	Prof. Dr.-Ing. Christian Dietrich

2024

Acknowledgments

I would like to thank all the people who constantly nagged me to finally finish this thesis.

I also would like to thank my IBR colleagues at TU Braunschweig, especially David, Signe and Stefan as well as my supervisor, Rüdiger.

Finally, I like to thank my parents, mostly because they wanted me to write this here.

The cover photo was generated using OpenAI's DALL-E.

List of publications

2021

- Nico Weichbrodt, Joshua Heinemann, Lennart Almstedt, Pierre-Louis Aublin and Rüdiger Kapitza. “Experience Paper: sgx-dl: Dynamic Loading and Hot-Patching for Secure Applications”. In *22th International Middleware Conference, Middleware’21*, `https://doi.org/10.1145/3464298.3476134`.
- Ines Messadi, Shivananda Neumann, Nico Weichbrodt, Lennart Almstedt, Mohammad Mahhouk and Rüdiger Kapitza. “Precursor: A Fast, Client-Centric and Trusted Key-Value Store using RDMA and Intel SGX”. In *22th International Middleware Conference, Middleware ’21*, `https://doi.org/10.1145/3464298.3476129`.
- Mohammad Mahhouk, Nico Weichbrodt and Rüdiger Kapitza. “SGXoMeter: Open and Modular Benchmarking for Intel SGX”. In *Proceedings of the 14th European Workshop on Systems Security, EuroSec ’21*, `https://doi.org/10.1145/3447852.3458722`.

2018

- Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler and Rüdiger Kapitza. “Troxy: Transparent Access to Byzantine Fault-Tolerant Systems”. In *Proceedings of the 48th International Conference on Dependable Systems and Networks, DSN’18*, `https://doi.org/10.1109/DSN.2018.00019`.
- David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Costa, Christof Fetzer, Pascal Felber, Peter Pietzuch and Rüdiger Kapitza. “EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution”. In *Proceedings of the 48th International Conference on Dependable Systems and Networks, DSN’18*, `https://doi.org/10.1109/DSN.2018.00048`.
- Vasily Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich and Rüdiger Kapitza. “STANlite-a database engine for secure data processing at rack-scale level”. In *Proceedings of the 2018 IEEE International Conference in Cloud Engineering, IC2E ’18*, `https://doi.org/10.1109/IC2E.2018.00024`.
- Nico Weichbrodt, Pierre-Louis Aublin and Rüdiger Kapitza. “sgx-perf: Performance Analysis Tool for Intel SGX Enclaves”. In *Proceedings of the 19th International Middleware Conference, Middleware ’18*, `https://doi.org/10.1145/3274808.3274824`.

2017

- Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In *Proceedings of the 26th USENIX Security Symposium, USENIX Security '17*, `https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-van_bulck.pdf`.

2016

- Stefan Brenner, Colin Wulf, Matthias Lorenz, Nico Weichbrodt, David Goltzsche, Christof Fetzer, Peter Pietzuch and Rüdiger Kapitza. "SecureKeeper: Confidential ZooKeeper using Intel SGX". In *Proceedings of the 17th International Middleware Conference, Middleware '16*, `https://doi.org/10.1145/2988336.2988350`.
- Nico Weichbrodt, Anil Kurmus, Peter Pietzuch and Rüdiger Kapitza. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves". In *Proceedings of the 21st European Symposium on Research in Computer Security, ESORICS '16*, `https://doi.org/10.1007/978-3-319-45744-4_22`.

Abstract

Trust is a fundamental requirement in distributed systems, especially in cloud computing environments. However, trust has been invalidated over the years through, e.g., bugs in the software used by the cloud provider or adversarial employees. A promising solution that aims to re-establish trust in cloud platforms is the use of a trusted execution environment (TEE), such as Intel's Software Guard Extensions (SGX). Intel SGX is one of the most sophisticated TEE implementations as it offers both confidentiality and integrity under a strong threat model. This makes SGX an attractive target for adversaries and several questions regarding its security and performance have been raised.

This thesis analyses the properties of SGX and identifies multiple performance issues as well as security weaknesses. Firstly, programming antipatterns in SGX-enabled software have been discovered, which, in combination with high secure execution mode transition costs, can lead to severe performance degradation. These antipatterns can be identified and fixed through the use of a toolkit presented herein. Secondly, this thesis uncovers a new class of application side-channel attacks called multithreaded controlled-channel attacks. This attack class has been ignored in the past but is a high-impact attack vector under the new SGX threat model as it can be used to gain code execution inside the TEE. Lastly, SGX's inflexible software deployment is an issue that previous work has tried to fix through the use of dynamic loading; however, these solutions lack support for hot-patching. In this thesis, the state of the art is extended with support for hot patching at runtime to remove costly TEE reloads.

Zusammenfassung

Vertrauen ist eine grundlegende Anforderung in verteilten Systemen, insbesondere in Cloud Computing-Umgebungen. Jedoch wurde das Vertrauen im Laufe der Jahre unter anderem durch Fehler in der von Cloud-Anbietern verwendeten Software oder durch böswillige Mitarbeiter untergraben. Eine vielversprechende Lösung, um das Vertrauen in Cloud-Plattformen wiederherzustellen, ist die Verwendung einer vertrauenswürdigen Ausführungsumgebung (Trusted Execution Environment, TEE), wie beispielsweise Intels Software Guard Extensions (SGX). Intel SGX ist eine der fortschrittlichsten TEE-Implementierungen, da sie sowohl Vertraulichkeit als auch Integrität unter einem starken Bedrohungsmodell bietet. Dies macht SGX zu einem lohnenswerten Ziel für Angreifer, und es wurden mehrere Fragen hinsichtlich der Sicherheit und Leistung von SGX aufgeworfen.

Diese Arbeit analysiert die Eigenschaften von SGX und identifiziert mehrere Leistungsprobleme sowie Sicherheitsschwachstellen. Erstens wurden Programmier-Anti-Pattern in SGX-fähiger Software entdeckt, die in Kombination mit hohen Kosten für den sicheren Ausführungsmodusübergang zu schwerwiegenden Leistungseinbußen führen können. Diese Anti-Pattern können mithilfe eines hier vorgestellten Toolkits identifiziert und behoben werden. Zweitens deckt diese Arbeit eine neue Klasse von Anwendungsseitenkanalangriffen auf, die als "multithreaded controlled-channel attacks" bezeichnet werden. Diese Angriffsklasse wurde in der Vergangenheit vernachlässigt, ist jedoch unter dem neuen SGX-Bedrohungsmodell ein hochwirksamer Angriffsvektor, da sie zum Einschleusen von Code in die TEE verwendet werden kann. Schließlich ist die unflexible Softwarebereitstellung von SGX ein Problem, das frühere Arbeiten durch die Verwendung von dynamischem Laden zu beheben versucht haben; diese Lösungen unterstützen jedoch kein Hot-Patching. In dieser Arbeit wird der Stand der Technik um die Unterstützung für Hot Patching zur Laufzeit erweitert, um kostspielige TEE-Neustarts zu vermeiden.

Contents

1 Introduction

In the last decade, usage of cloud resources has increased drastically [73, 62]. What started with the offering of (theoretically) limitless virtual machines, expanded into more and more specialized offerings meant to satisfy every use-case, prompting businesses to move away from self-hosted infrastructure to a cloud provider of their choosing. However, moving to the cloud can be problematic for certain businesses as these handle and process sensitive data, e.g., Personally Identifiable Information (PII), medical data or financial data. Should such businesses use the cloud, they have to hand over the data and trust that the cloud provider and its personnel is not stealing or modifying the data.

Encryption can protect data at rest, and is even mandated for such sensitive data handled by companies inside the European Union (EU) as per the General Data Protection Regulation (GDPR), Art. 32. When processing the encrypted data, however, there is an issue: The data either (i) needs to be decrypted or (ii) the processing has to work on encrypted data. Decryption is not an option as again the sensitive data would be available to the cloud provider. Processing encrypted data has been a field of active research for many years under the name *homomorphic encryption* [64]. While homomorphic encryption is practical for some basic operations, it suffers from very low performance for other, more complex operations to not being able to do some operations at all, depending on the scheme used [1]. In 2015, a new, third option became available. Intel unveiled its Software Guard Extensions (SGX) technology that claimed secure computation of sensitive data with complete isolation from the hardware provider [60]. The only trust needed would be in Intel to implement the system correctly. What Intel announced is commonly known as a Trusted Execution Environment (TEE). With TEEs, data can be safely decrypted for processing as such data is still protected from access through a multitude of techniques. The concept of TEEs is not novel and has been used in ARM processors, mainly in the mobile processor market, in the past [7]. There, the implementation was called TrustZone which enabled ARM processors to switch between a *normal* and a *secure world*. Samsung [74] utilizes TrustZone to enable a secure and trusted boot procedure for Android in addition to integrity verification of the running Operating System (OS). On desktop and server platforms, the first steps towards trusted computing were done with the utilization of Trusted Platform Modules (TPMs) [51] and later Intel Trusted Execution Technology (TXT) [48] which provided similar functionality.

TEEs are, however, not perfect and have drawbacks. In the case of Intel SGX, the TEE is a special execution mode of the processor and switching between this mode and normal execution is non-trivial with a performance cost. It is also not possible to just run unmodified legacy software in this mode. New software either needs to written specifically for SGX or legacy software needs to be ported to SGX. When doing so, developers need to be aware of the intricacies of SGX to write secure and *performant* software. This is even more important if SGX-enabled software should be *updatable* during execution, a property that is often used in cloud deployments.

Lastly, every system that claims it is secure attracts security researchers proving the opposite. This is also true for Intel SGX whose availability caused a lot of side-channel attacks to be discovered that were partly applicable to other mechanisms inside the processor as well. The severity of these attacks vary from arbitrary code execution [17] over data exfiltration [93] to denial of service [52] and depend on the SGX properties used.

The research presented in this thesis focuses on the topic of performant and updatable SGX software and attacks against such software. SGX was chosen as it is the most available technology that also offers the most features. The techniques described herein can be adapted to other TEEs with similar features and do not require SGX specifically. The main requirement is that applications are partitioned with the secure part running inside the TEE. This can be achieved through the use of a platform-agnostic TEE framework that follows this model, e.g. Google Asylo [11], or by building an enclave-like abstraction layer on top of another TEE technology.

1.1 Publications

The article thesis at hand makes several contributions regarding the security and usage of Intel SGX. The individual contributions are contained in the following three scientific publications in conference proceedings. These three publications are contained in this document and are listed (with their respective page numbers) in the following:

1. Nico Weichbrodt, Pierre-Louis Aublin and Rüdiger Kapitza. "sgx-perf: Performance Analysis Tool for Intel SGX Enclaves". In: 19th International Middleware Conference Proceedings (2018), pp. 201–213 (on page 23)

2. Nico Weichbrodt, Anil Kurmus, Peter Pietzuch and Rüdiger Kapitza. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves". In: Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS, 2016), pp. 440–457 (on page 43)

3. Nico Weichbrodt, Joshua Heinemann, Lennart Almstedt, Pierre-Louis Aublin and Rüdiger Kapitza, "Experience Paper: sgx-dl: Dynamic Loading and Hot-Patching for Secure Applications". In: 22nd International Middleware Conference Proceedings (2021), pp. 91–103 (on page 67)

1.2 Contributions

The contributions of this article thesis in general and of the research papers stated in the previous section in particular can be attributed to the following research challenges:

Research Challenge 1: Detecting and Fixing Suboptimal Enclave Interfaces

With Intel SGX a TEE implementation exists that is available in commodity hardware to the general public. While its programming model is similar to traditional applications, it differs in key areas that, through inexperience or simply wrong usage, can cause performance of security sensitive application to be lower than expected [22, 9]. The first contribution aims to alleviate this issue. sgx-perf is a high-level logger and analyser that helps developers write better code by giving concrete change recommendations. The application is first run normally with the sgx-perf logger attached which records relevant data into a profile. With the sgx-perf analyser, the profile of an application can then be analysed for performance bottlenecks. The sgx-perf analyser gives concrete recommendations on changes that will benefit enclave performance. An evaluation showing the effectiveness of sgx-perf is provided.

Research Challenge 2: Attacking and Defending Multithreaded Enclaves

As Intel SGX is commonly available, it has become a prominent target of security researchers who try to break its security guarantees. While no-one has broken SGX directly, a steadily growing number of side-channel vulnerabilities and micro-architectural attacks has been found and published. The second contribution, AsyncShock, introduces a novel side-channel attack vector on multithreaded SGX enclaves. AsyncShock enables an attacker to stop and resume enclaves threads at will and allows them to force the enclave into a specific thread schedule. In enclaves that contain synchronization bugs, this feature can be used to reliably trigger such a bug which in turn can enable further malicious actions, from simple secret exfiltration up to control flow hijacking. AsyncShock was successfully used against two different bugs and was able to exfiltrate sensitive enclave data.

Research Challenge 3: Enabling Secure Modularity in Enclavized Applications

Intel SGX has been the TEE implementation of choice when porting existing applications to run inside a TEE. Certain system software and capabilities, however, have been missing so far, notably support for extending an application at runtime. The third contribution is sgx-dl, a framework that allows for securely adding and removing dynamic code from an SGX enclave at runtime. sgx-dl enables enclave developers to update and extend enclaves after launch by embedding a dynamic loader inside the enclave. With this dynamic loader, enclave can be built in a modular fashion that also allows for updates of the modules over the enclave's lifetime. sgx-dl has a low overhead and is shown to work with different approaches of integration, from only dynamically loading a single part up to dynamically loading a whole application.

1.3 Outline

This extended overview of the article thesis is meant to be a summary of the publications presented in Sections 3, 4 and 5. A full list of publications by the author is given on Page v of this document. Those publications that are considered as an essential part of this work are contained in the thesis and are included in their respective sections. The thesis makes explicit references to those papers to highlight their context and the relation between the publications. This thesis is structured as follows: Section 2 gives an overview over the TEE implementation Intel SGX as the presented works are built on top of it. In Section 3, the performance profiling toolkit sgx-perf is presented. It enables developers to profile an unmodified Intel SGX application and gather performance statistics regarding enclave transitions. sgx-perf can give recommendations based on the gathered data helping developers to optimize their application and prevent performance bottlenecks. In Section 4, the AsyncShock attack system is presented. With AsyncShock it is possible to control enclave threading such that synchronization issues in SGX enclaves can reliably be exploited. The dynamic loading and hot-patching framework sgx-dl is discussed in Section 5. It offers a novel way to add dynamic loading and hot-patching capabilities to Intel SGX enclaves with very little overhead. Finally, a summary of this extended overview and a short outlook on future work is given in Section 6.

2 Background

In the last years, Trusted Execution Environments (TEEs) have become commonly available in computing systems, the most common implementation being ARM TrustZone [7] due to its prevalence in mobile devices such as smartphones. TrustZone implements two execution contexts, the *normal world* and *secure world*, with the secure world being isolated against access from the normal world but not vice versa. It is typically used to separate the untrusted mobile operating system in the normal world from other telephony related software in the secure world. While TrustZone is the de-facto standard on mobile devices, traditional desktop and server PC systems were lacking a viable option. In 2007 [48] Intel released Intel Trusted Execution Technology (TXT) [48, 35] which aims at attesting the authenticity of the platform and operating system while also building a chain of trust for software started later on the system. Intel TXT requires a TPM as a root of trust from which the chain of trust is bootstrapped.

While TXT tries to authenticate the whole system step-by-step, another approach is to create a trusted environment for an application from scratch on an otherwise untrusted platform. In 2015 the first Intel processors with support for Intel Software Guard Extensions (SGX) were released which does exactly that. Similarly, AMD released their competing implementation, AMD Secure Memory Encryption (SME) and AMD Secure Encrypted Virtulization (SEV), in 2016 [2] and extended it in 2020 with Secure Nested Paging (SNP) [3]. Since this thesis presents work based on SGX, the following section will explain it in detail. However, the concepts of the publications presented in this thesis can be adapted to work on other TEE implementations.

2.1 Intel Software Guard Extensions (SGX)

Intel SGX is an extension of Intel's x86 architecture and implements a trusted execution environment that achieves confidentiality and integrity during execution [37]. Its architecture differs from the previously mentioned TEE implementations as SGX is neither isolating on a OS nor Virtual Machine (VM) level. SGX instead allows the creation of so-called *enclaves* which are isolated compartments that are part of processes and therefore isolate at the application level. This also means, that enclaves always run in user mode, the lowest possible privilege level in x86. A process can launch multiple enclaves that are also isolated from each other.

From an application perspective, the enclave is nothing more than a special memory area. This memory, however, is not directly accessible from the untrusted application; it can neither be read from nor written or jumped to. Instead, new instructions are used to set up and manage enclave memory. SGX adds two new instructions, `ENCLU` and `ENCLS`,

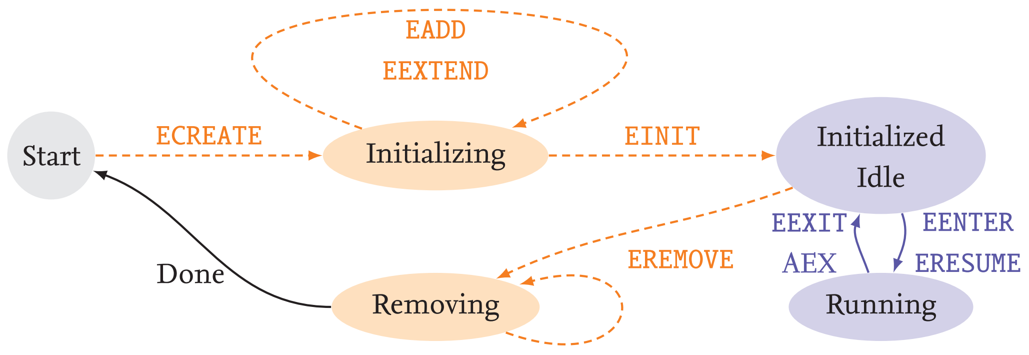


Figure 2.1: Lifecycle of an SGX enclave.

with several sub-instructions accessed through specific values in the `rax` register. Different responsibilities of the enclave lifecycle are given to user- and kernel mode which is why `ENCLU` is only available to user mode and `ENCLS` is only available to kernel mode. For example, creation of the enclave is handled by the kernel but entering and exiting an enclave is only possible to user mode.

The security of SGX enclaves are based on a single root of trust: Intel. To verify whether an enclave is running on real Intel SGX hardware, the platform can generate a *quote* for an enclave which is a cryptographic proof signed by the platform and checked by an external service run by Intel that certifies the validity of the platform and correct initialization of the enclave [5]. Intel assures that privacy is maintained by utilizing Intel Enhanced Privacy ID (EPID). EPID is a group signature scheme that makes it possible for a platform to sign quotes without identifying each signer or linking signatures to a single signer [23]. In this case, with the signer being the processor itself, EPID makes sure the identity of the machine and its owner stays private. On newer platforms (Ice Lake-SP and up), EPID has been replaced by *flexible launch control* [4] which allows the platform owner to supply their own quote generation and verification infrastructure [50].

Verifying the platform is only one part with the platform verifying the enclave being the other. During enclave creation, a rolling SHA256 hashsum over all enclave pages is calculated. This hashsum is called the *measurement* of the enclave. Together with the expected and signed measurement supplied by the enclave developer it is possible to check that the correct and unaltered enclave has been launched. This mechanism therefore ensures trust in the platform being genuine and the launched enclave being unmodified.

Enclave Lifecycle

Enclave creation is handled by the kernel and uses the `ENCLS` sub-instructions `ECREATE`, `EADD`, `EEXTEND` and `EINIT`. Kernel instructions are shown in Figure 2.1 with the orange dashed arrows. First, `ECREATE` is used to create a new empty enclave. With `EADD` the code and data pages of the enclave are added sequentially. During this phase, `EEXTEND` is used to advance the calculation of the measurement. Finally, `EINIT` initializes the enclave. At this step, the calculated measurement is compared against a supplied one to ensure that (i) the correct enclave has been launched and (ii) the enclave has not been altered. Additionally, a valid *launch token* has to be presented which can only be generated by the *launch enclave*, see Section 2.1.1.

After enclave initialization, execution can begin. Untrusted code, however, is not able to just jump to enclave code. Instead, the `ENCLU` sub-instructions `EENTER`, `ERESUME` and `EEXIT` have to be used to transition to and from enclave code. As `ENCLU` is only available to user mode, the enclave can only be entered from unprivileged code. This is shown in Figure 2.1 with the blue solid arrows. With `EENTER` the enclave can be entered and execution of trusted code will start. The enclave can be left with `EEXIT`. `ERESUME` is used to re-enter an enclave after an interrupt has caused an Asynchronous Exit (AEX).

An AEX is performed when an interrupt or exception occurs during enclave execution. In that case, the current enclave state is saved in enclave memory (see Section 2.1.1) and the enclave is automatically exited to handle the interruption. Afterwards, execution control is transferred to the Asynchronous Exit Pointer (AEP), a pointer to a handler function specified during `EENTER`, that chooses whether to resume enclave execution at the interruption point with `ERESUME`. The AEP handler can also perform other operations first or choose to re-enter the enclave anew with `EENTER`. As the AEP handler is untrusted code, the enclave must not expect a certain action and must instead be capable of handling resumption and restart of the execution. With the newest SGX extension, AEX-Notify [41], however, the enclave can force `EENTER` semantics even when `ERESUME` is used (see Sections 2.1.2 and 4.2).

When the enclave is no longer needed, it can be removed from memory by the kernel using the `EREMOVE` sub-instruction. The kernel has to remove every enclave page individually using that sub-instruction. As a design quirk, removing a page from an enclave does not prevent one from entering the enclave again. According to the instruction reference [40], removing a page does not invalidate an enclave's initialized status and therefore would allow entering it with missing pages. This is actually unproblematic, as long as there are no memory accesses to these missing pages as they would cause a fault. This potential transition has been omitted from Figure 2.1 as instructing the Software Development Kit (SDK) (see Section 2.2) to remove an enclave will remove all pages of that enclave in one call and prevent the enclave from being entered and is therefore not used in practice.

2.1.1 Technical Details

In the following, multiple technical details of SGX are presented that are needed for the overarching understanding of the works presented in this thesis. Further details are presented in their respective papers when needed.

Launch Enclave

Technically, SGX is only capable of launching enclaves that are signed by a specific key held by Intel (excluding debug enclaves) which results in a problem: Intel would need to sign all enclaves with the same key. This issue is remedied by having a special *launch enclave* which can generate *launch tokens*. Launch tokens are bound to the measurement of the enclave that should be launched. On enclave launch, the matching launch token is also presented to SGX and verified before launch. This mechanism makes it possible to have different signing keys, e.g., one per organization developing enclaves, and the launch enclave only needs to know about those. On newer platforms (Gemini Lake, Ice Lake-U and newer), system firmware can set the measurement of the launch enclave through special registers at boot time to allow third-party launch enclaves with potentially different validation mechanisms. This feature was not available on the first SGX-capable platforms but is now widely supported. Its existence does not change the results presented in this thesis, it merely removes Intel even more from the Trusted Computing Base (TCB) by allowing users to fully control enclave launches.

Secure Enclave Memory

All enclaves reside in a special memory area that is encrypted and integrity protected at all times. This memory area is called the Enclave Page Cache (EPC). The EPC is part of a larger special memory area, the Processor Reserved Memory (PRM). The PRM is configured by the system firmware at boot time. On first generation systems it has a maximum size of 128 MiB and on newer systems a size of 256 MiB. The EPC occupies the most part of the PRM with 93 MiB and 188 MiB, respectively. The mentioned EPC sizes are small when compared to today's DRAM sizes. Enclaves that are bigger than the EPC are supported as SGX offers a paging mechanism that can swap the content of EPC pages to untrusted memory, see next subsection.

Other parts of the PRM are used for enclave metadata, the Enclave Page Cache Map (EPCM), and for storing the hash-tree that integrity protects the PRM [36]. The EPCM is a data structure that holds metadata for every page inside the EPC. This metadata includes the enclave the page belongs to and the page permissions. The EPC contains the actual enclave code, data and management structures.

Systems with higher usable EPC [75] have been available since 2021 exclusively on the Ice Lake-SP platform. Here, the EPC size is no longer fixed but rather dynamically allocated and deallocated when needed up to a configurable maximum share of system memory.

EPC Paging

To enable the creation and use of enclave that are larger than the size of the EPC, SGX offers a paging mechanism. The paging mechanism ensures confidentiality and integrity by re-encrypting the page before page-out and ensures freshness by keeping the nonce used for encryption in the EPC itself in a different page. One such EPC page can store 512 nonces (8 bytes per nonce, 4 KB page size) so one EPC page can be used to track this many paged out EPC pages. Such a page can itself also be paged out. Developers can ignore this paging mechanism as it is transparent to the enclave. As pages need to be encrypted and decrypted, however, it is a costly mechanism that should be avoided by making sure all required enclave memory can fit inside the EPC. In general, the performance of SGX enclaves is the topic of *sgx-perf*, the first paper presented in this thesis.

On Ice Lake-SP systems, this page mechanism is still available with the aforementioned performance cost but will usually be not used as the flexible allocation of EPC pages make paging obsolete except when both enclaves and untrusted applications both need a substantial amount of system memory.

Enclave Data Structures

Each enclave comprises not only its code and data but also several management data structures which are created at enclave launch. Most importantly, each enclave contains a SGX Enclave Control Store (SECS) which holds enclave metadata such as the enclave measurement. The SECS also specifies the size and memory range of the enclave. As enclaves cannot be jumped to, they require definition of entry points. These entry points are directly coupled to threads. Each enclave contains at least one Thread Control Structure (TCS) which specifies for one thread where the enclave should be entered and where the State Save Area (SSA) for this TCS is located. The SSA is the memory area where registers are saved in case of an AEX. Each additional thread that should enter an enclave therefore requires its own TCS and SSA.

2.1.2 Side-Channel Attacks

SGX promises high security under a permissive threat model: It is assumed, that an attacker has administrator-like privileges and physical access to the machine. The host operating system is not trusted as well as a potential hypervisor. This threat model makes SGX a prime target for security researchers which have found a number of flaws and defects in the interaction of SGX and other processor sub-systems ranging from data disclosure to control flow hijacking.

Before highlighting these works, we need to distinguish between flaws in the SGX subsystem and its implementation as well as the interaction of SGX with other systems. To my knowledge, no security relevant flaw directly inside SGX has been found so far. All security relevant flaws are contained to interactions with different parts of the processor, e.g., the cache or the Memory Management Unit (MMU), in the form of side-channels. Most attacks also not only affect SGX but also break the VM isolation. Intel has also stated that SGX was not designed to be resilient against side-channel attacks [42] and it's on the enclave developer to write software resilient to side channel attacks. Multiple techniques that claim side-channel attack resiliency in SGX have been proposed, such as Varys [67], MoLE [56] and others [26, 38, 19]. There also exist some applications using SGX while also claiming side-channel resiliency, such as TRUSTORE [66].

Not all side-channel attacks are the same. The two main categories of side channel attacks are *microarchitectural* and *application* side-channel attacks. Microarchitectural side-channels exist due to some flaw in the architecture of the processor whereas application side-channels exists due to a flaw in the architecture of the application. The first microarchitectural side-channel attack against SGX is Spectre [53] which abuses the speculative execution of the processor to leak sensitive data. Spectre laid the groundwork for many more microarchitectural side-channel attacks such as SgxPectre [27], Foreshadow [91], ZombieLoad [78], Plundervolt [63], CacheOut [76] or LVI [92]. Most of the underlying flaws have either been fixed by Intel through microcode updates or software mitigations exist.

Application side-channel also exist, however, less research has been published as they mostly focus on one specific application or application type. One of the first papers that looks at application side-channels is *AsyncShock*, the second paper presented in this thesis. Another prominent example is presented in our paper *Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution* [93] in which we showed that through observation of page table accesses it is possible to extract the secret session keys used in EdDSA from an enclavized `libgcrypt`. Here, operating on the key one bit at a time, a binary `1` causes slightly different timing from operating on binary `0` which is enough information leakage to recover the secret key outside the enclave by just observing the page table. The same approach was again used to build *SGX-Step* [90], a generic framework that allows single stepping through an enclave by continuously interrupting it. A defence against SGX-Step has been released as an SGX extension named *AEX-Notify* [29, 41], see Section 4.2.

SGXv2 Extensions and Improvements

In October 2014 the second version of SGX has been specified by Intel[43] with first processors becoming available at the end of 2017[1]. The main advancement of SGXv2 is the possibility to add enclave pages after enclave creation as well as the capability to change enclave page permissions at runtime. With SGXv1, the memory layout of the enclave is fixed after initialization. It is neither possible to add or remove pages nor can the page permissions be altered. This prohibits enclaves from securely loading code dynamically after creation as this feature would require enclave pages to be readable, writable and executable at the same time. Pages with such permissions are a security issue [15]. Furthermore, an enclave needs to know its peak memory usage and has to load all pages necessary to support that at creation time. If peak memory usage is significantly higher than normal then this increases enclave start up time.

Now, with SGXv2, the enclave start can be accelerated as the enclave can be launched

[1]`https://www.intel.com/content/www/us/en/support/articles/000058764/software/intel-security-products.html` lists Gemini Lake which launched in Q4 2017.

with a minimal amount of pages. When more pages are needed, these can be added after enclave initialization. This feature can for example be used to implement dynamically sized heaps and stacks and reduces enclave start time. Similarly, SGXv2 enables enclave page permission changes after initialization. Together with adding pages at runtime, it is now possible to securely load code as pages can be made non-executable for loading and non-writable for executing. SGXv2 enabled the work on *sgx-dl*, the third paper presented in this thesis.

2.2 The Intel SGX Software Development Kit

To make software development easier, Intel provides a SDK since September 2016 with support for SGXv2 since February 2018 [45]. The SDK allows enclave developers to write software with enclaves or integrate enclaves into existing software without the need to understand or use the SGX specific instructions. This is important as it makes adoption of the technology by developers easier and faster.

Beside the official SDK by Intel, there also exist other SDKs, e.g., Asylo [11] by Google and Teaclave [6] originally by Baidu which has been continued by the Apache Software Foundation. Both offer generic support for building enclavized applications and are not dependent on one TEE technology but rather can work with multiple. For the work presented here, the official Intel SGX SDK was used due to its availability and maturity. Both Asylo and Teaclave were either not available or not in a usable state, e.g. due to missing features, during the work on the presented projects.

With the Intel SGX SDK, a partitioning approach to application development has been chosen, i.e., the application consists of an untrusted part and one or more enclaves which are responsible for security sensitive executions. To make it easy to call into enclaves, the complex system of enclave transitions are hidden from developers through two mechanisms called ECalls (enclave calls) and OCalls (outside calls). For developers, ECalls and OCalls look and behave like normal functions calls, however, internally enclave transitions are made using the aforementioned instructions.

The SDK is split into three parts. The first part is the Linux kernel driver[2] which handles enclave creation, destruction and all enclave memory management. The second part is the Platform SoftWare (PSW)[3] which is a bundle of shared libraries needed to execute SDK applications as well as the launch and quoting enclave from Intel. The last part is the actual SDK which consists of special in-enclave standard C and C++ libraries, a wrapper code generator and other helpful static libraries.

Linux Kernel Driver The Linux kernel driver is responsible for creating and destroying enclaves and offers an `ioctl` interface through the `/dev/isgx` device. It also handles enclave memory over-subscription through the use of a page swapping mechanism. If

[2]`https://github.com/intel/linux-sgx-driver`

[3]Both the PSW and SDK are available at `https://github.com/intel/linux-sgx`

more virtual enclave memory is needed than EPC is available, the driver will move pages into untrusted memory to free up EPC space. These pages are re-encrypted and page metadata, such as a hashsum over the content, is stored in EPC.

Platform SoftWare The PSW contains all needed shared libraries to run applications built with the SDK. This is mainly the Untrusted RunTime System (URTS), but also the libraries needed to communicate with the architectural enclaves. Additionally, the PSW supplies the architectural enclaves needed for execution: the launch enclave for generating launch tokens and the quoting enclave for generating quotes for remote attestation.

SDK To actually develop applications with enclaves, the SDK is required. It provides a stripped down version of the C and C++ standard libraries as an enclave needs its own self-contained standard library. These libraries do not do any system calls and therefore do not contain any functions that would need to do system calls. The bundled C standard library does, however, support synchronization primitives like mutexes that will leave the enclave, see Section 3.1.

For dispatching ECalls and OCalls, the Trusted RunTime System (TRTS) is included. Other libraries included with the SDK are a cryptography library (`tcrypto`), a library that handles sealing of data (`tservice`) and a library that provides a filesystem-like Application Programming Interface (API) inside the enclave (`tprotected_fs`) Finally, the SDK contains the wrapper code generator `edger8r` which is responsible for generating code for the enclave interface for both the trusted and untrusted side. The wrapper code is needed as the SDK wants to make enclave transitions seem like function calls. Essentially, the wrapper code needs to marshall and unmarshall the function call arguments before and after transitioning from non-enclave to enclave mode and vice-versa.

With the SDK, enclave code looks very similar to normal C code. As seen in Listing 1 on Page 15, two files are present. The `app.c` contains the untrusted application code which initializes the enclave and performs an ECall. The `enclave.c` contains the trusted ECall implementation which performs some secure computation. Comparing line 18 and 30 uncovers that the function signature of the ECall is different inside and outside the enclave. This is due to the wrapper code generator generating slightly different code for inside and outside as it has to account for two additional properties. First, the ECall/OCall might fail which necessitates a status return value and in turn moves the function return value to a parameter. Second, there might be multiple active instances of the same enclave which necessitates the addition of an identification parameter (enclave ID, or `eid`).

2.2.1 The Enclave Description Language

To describe the enclave interface, the Enclave Description Language (EDL) is used. Developers describe all the ECalls and OCalls of their enclaves in EDL files which are read by the wrapper code generator to create the trusted and untrusted wrapper functions.

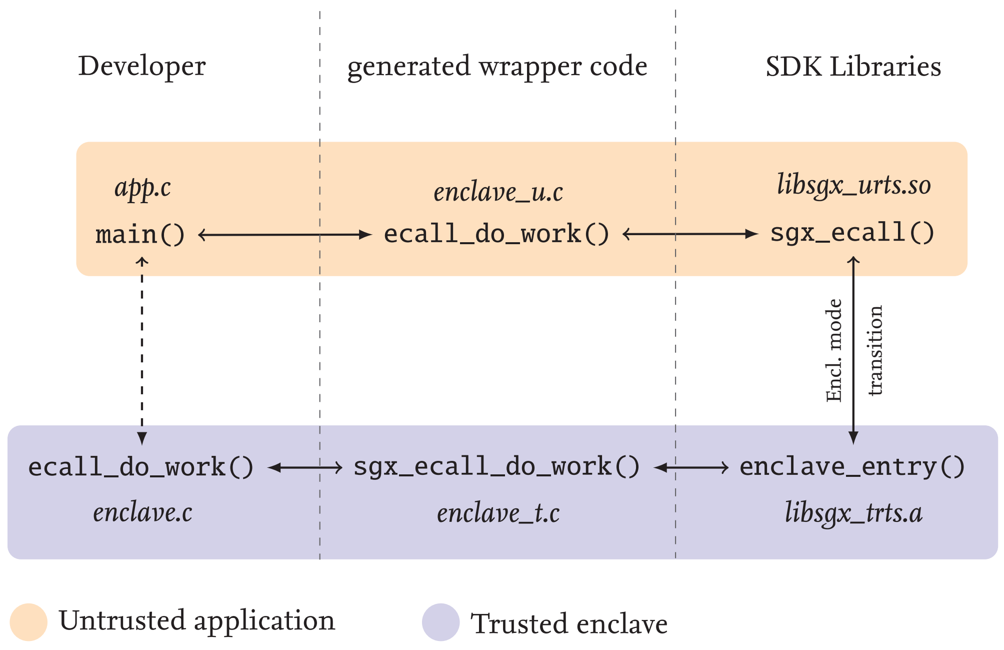


Figure 2.2: ECall control flow in the Intel SGX SDK.

Listing 2 on Page 15 shows an example for a minimal EDL file with one ECall. The EDL simply lists C function signatures of the available ECalls. More details can be found the Developer Reference of the SDK [45].

2.2.2 ECall Dispatching

Figure 2.2 shows how ECalls are handled inside SDK applications. The generated wrapper code defines all the ECalls that have been previously defined via the EDL. Its purpose is to marshall all arguments into one `struct` that is then passed to the URTS together with a unique numeric identifier that identifies this ECall. Note that not this specific call instance is identified by the identifier but the ECall itself. The URTS then sets up the processor state and performs an enclave transition to the TRTS.

The TRTS is mainly responsible for dispatching ECalls inside the enclave. It does so by first checking whether the supplied ECall identifier is valid for this enclave. If so, it calls the trusted part of the generated wrapper functions which then unmarshalls the argument and calls the actual ECall implementation. After the ECall returns, a possible return value is passed through the wrapper code to the TRTS, URTS and wrapper again back to the calling application.

As can be seen, this call is not simply a function call even if it looks like it from a developer's perspective. The enclave and SDK transitions add time to the execution which makes ECalls slower when compared to function calls.

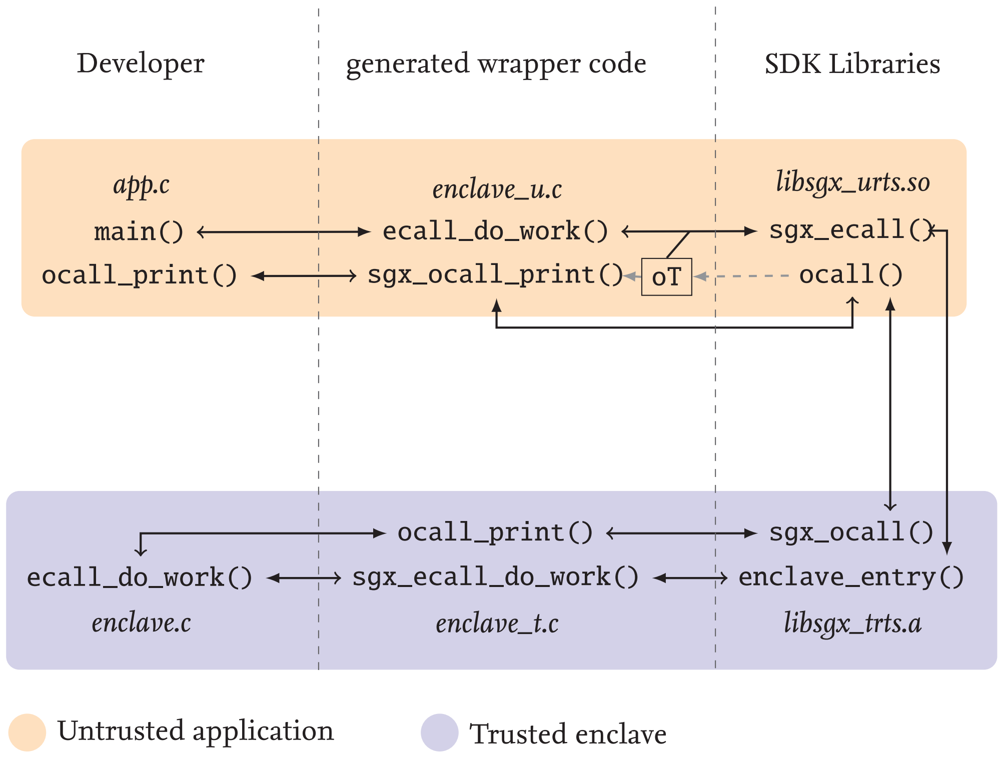


Figure 2.3: OCall control flow in the Intel SGX SDK.

2.2.3 OCall Dispatching

OCalls are the reverse of ECalls and therefore work similarly as shown in Figure 2.3. When the enclave wants to execute an OCall, the in-enclave wrapper code is called which marshalls the call arguments before calling into the TRTS. The TRTS leaves the enclave to call into the URTS which then has to call the untrusted wrapper code. The URTS, however, needs knowledge about the existing OCalls as it is linked dynamically into the application. For this, the application prepares a special *OCall table* (oT in Figure 2.3) that is passed to the URTS with every ECall. Using the table, the URTS can get the function pointer to the correct wrapper functions and call it. Lastly, the wrapper calls the actual OCall implementation, collects an optional return value and passes it back into the enclave.

The ECall and OCall system is an elegant way to abstract the complicated mechanism of switching between enclave and non-enclave mode away from developers. From a developer's perspective, the enclave boundary is crossed via seemingly simple function calls. However, these abstractions come with potentially unexpected performance drawbacks which are explored in the following chapter.

```c
/***** app.c *****/
#include <sgx_urts.h>
#include "enclave_u.h"  // generated header

int main() {
    sgx_launch_token = {};
    int updated = 0;
    sgx_enclave_id_t eid = 0;
    sgx_status_t ret = sgx_create_enclave("enclave.signed.so", 1,
        &token, &updated, &eid, NULL);
    if (ret != SGX_SUCCESS) {
        exit(1);
    }

    int result = 0;
    ret = my_add_ecall(eid, &result, 2, 7);
    if (ret != SGX_SUCCESS) {
        exit(2);
    }

    printf("2 + 7 = %d\n", result);
}
/***********************************************/
/***** enclave.c *****/
#include "enclave_t.h"  // generated header

int my_add_ecall(int a, int b) {
    return a + b;
}
```

Listing 1: SGX SDK code example

```
enclave {
    trusted {
        public int my_add_ecall(int a, int b);
    };
};
```

Listing 2: EDL example

3 On the Performance of Utilizing Enclaves

With the SGX SDK, Intel offers an easy way for developers to develop enclave applications and coupled with SGX's availability in commodity hardware, it seems to be an obvious choice for developers to use. When SGX was first introduced to the public, micro [98] and macro benchmarks [22] were performed to get an understanding of the performance characteristics of the new technology and two mechanism were quickly identified as potential performance bottlenecks.

As shown in Section 2.2, the SGX SDK offers an intuitive way for implementing calls into an enclave that mimic function calls. However, while this mechanism is nice from a developer's perspective, it is the fact that a transition into the TEE has to be made that makes this abstraction problematic. With modern compilers and hardware, function calls are fast with little overhead as they comprise only a few instructions and are not causing a context switch. Developers using the offered function call abstraction on the SGX SDK might have the same expectation. Weisse et al. [98] showed in 2017 that this expectation is false. They measured enclave transition times for SDK ECalls and OCalls and showed that these transitions require 8,600 to 14,000 cycles of processing time to execute. Their methodology, however, is not perfect. First, the measured numbers are round-trip times, so they include two transitions. Second, the round trip includes all SDK functionality needed to make the transition. The measured numbers are therefore not only the hardware transition time, but also the SDK processing time.

This raises more questions: Are transitions into the enclave less, equal to, or more expensive than transitions out of the enclave? And how much of the time is spent inside the SDK itself and, in light of the recent side channel attack mitigations, do these mitigations have an effect on the transition times?

The other identified performance bottleneck is the size of the EPC and the resulting page swapping for enclaves that do not fit into the EPC. With only 93 MiB (and later 188 MiB) the EPC is large enough to hold code but too small to hold data of more sophisticated applications. The small EPC size was an architectural limitation and has been removed in newer iterations of SGX which makes this almost a non-issue for current enclave development. Only in cases where the enclave has a working set larger than the system memory, EPC swapping will still be an issue as well a normal page swapping due to memory exhaustion on the host. Therefore, knowing an enclaves working set is still relevant when looking at big enclavized software, such as Database Management Systems (DBMSs). The enclave working set is also relevant because the EPC is encrypted and therefore executing enclave code and loading and storing data inside the EPC requires the processor to decrypt and encrypt EPC pages. In SCONE [9] it has been shown that as soon as the data does not fit into the processor's caches any more (but still inside EPC), the encryption can cause a 10x overhead compared to non-SGX execution.

In the end, enclave transition times and enclave working set size are both metrics enclave developers need to be aware of and need to take into account when developing enclavized software.

3.1 sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves

In *sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves* [95] (on page 23), we present a system that helps enclave developers by gathering and analysing these metrics. sgx-perf is unique in that it is giving recommendations to developers based on the gathered metrics on how to improve enclave performance by structuring their code differently.

First, however, the transition time measurement from Weisse et al. was verified, but with a modified SGX SDK. Instead of measuring the time before and after the ECall, in sgx-perf, the SDK was itself modified to measure directly before the instruction responsible for the enclave transition. Similarly, inside the enclave, the first instruction executed immediately transitions back out of the enclave to the modified SDK where the second measurement was made. This still measured a round-trip, but excluded the SDK to gauge its effect on the transition time. Measuring a single transition and not a round-trip was actually not possible at time as the `rdtscp` instruction for reading the timestamp counter of the processor caused it to fault when inside an enclave. This behaviour was not intended by Intel and has been fixed in SGXv2 which was not available at the time. Furthermore, the measurement has been repeated with multiple μcode versions to gauge the effect of side-channel mitigations on the transition times.

For this thesis, the measurement has been repeated to include μcode versions that were released after the publishing of sgx-perf. Tables 3.1 and 3.2 show the measured transition times for different μcode versions on two machines. The first machine only offers SGXv1 and therefore all times are round trip times. The second machine offers SGXv2 which allows measuring entering and exiting the enclave separately.

As can be seen, multiple code updates over the course of the machine's lifespan had a significant impact on enclave transition performance. While the updates are optional in the sense that there is no requirement to install them to keep using SGX, they actually are mandatory if running enclaves should be attested by the Intel attestation service. SGX embeds a Security Version Number (SVN) into the attestation report that serves as a version number of the SGX subsystem. To enforce that μcode updates with mitigations against attacks are actually installed, the attestation service is refusing to attest enclaves with a SVN that is too old. In the end, μcode updates will be installed which on one hand restores

µcode Version	Cycles	Time [µs]	Increase
`0x56` (Stock)	6531 ± 740	≈ 1.92	-
`0xBA` (No security updates)	6468 ± 752	≈ 1.90	$\approx 0.99\times$
`0xC2` (Spectre)	11493 ± 752	≈ 3.38	$\approx 1.76\times$
`0xC6` (Foreshadow)	14308 ± 1227	≈ 4.21	$\approx 2.19\times$
`0xCC` (MDS Attacks)	15821 ± 1303	≈ 4.65	$\approx 2.42\times$
`0xD4` (MDS Attacks)	16273 ± 1415	≈ 4.79	$\approx 2.49\times$
`0xD6` (No security updates)	16516 ± 1430	≈ 4.86	$\approx 2.53\times$
`0xDC` (CacheOut / SGAxe)	15711 ± 1321	≈ 4.62	$\approx 2.41\times$
`0xE2` (Plundervolt)	15497 ± 1329	≈ 4.56	$\approx 2.37\times$

Table 3.1: Enclave transition times (round trip) with standard deviation for different µcode versions on a SGXv1 machine (Xeon E3-1230v5). For each µcode version the mitigated attacks are noted in parentheses. Increase is always compared to the *Stock* baseline.

µcode Version	Cycles (enter/exit)	Time [µs]	Increase
`0x2E` (Stock)	2560 ± 1276 / 1180 ± 407	$\approx 1.97/0.91$	-
`0x46` (MDS Attacks)	4939 ± 1622 / 1195 ± 398	$\approx 3.80/0.92$	$\approx 1.93\times/1.01\times$
`0x78` (CacheOut / SGAxe)	4697 ± 1606 / 1198 ± 402	$\approx 3.61/0.92$	$\approx 1.83\times/1.01\times$
`0xA0` (Plundervolt)	4902 ± 1629 / 1203 ± 404	$\approx 3.77/0.92$	$\approx 1.91\times/1.02\times$

Table 3.2: Enclave transition times with standard deviation for different µcode versions on a SGXv2 machine (Core i7-1065G7). For each µcode version the mitigated attacks are noted in parentheses. Increase is always compared to the *Stock* baseline. Enclave enter and exit measured separately.

security by mitigating attacks but on the other hand reduces performance due to increased enclave transition times.

All this reinforces the main point of sgx-perf: Enclave transitions are expensive and should be avoided when possible. As mentioned earlier, the SGX SDK hides enclave transitions as function calls which can cause performance issues if developers are not careful. At the time of publishing, the only tooling with regard to analysing SGX performance that existed was the Intel VTune low-level profiler [49]. The profiler is great to find performance bottlenecks in, e.g., single functions, hot loops and similar. There is, however, no analysis of the enclave interface. With sgx-perf this gap in the tooling has been closed as it is the first tool that not only analyses the enclave interface but also gives recommendations on how to change it to improve performance.

sgx-perf consists mainly of two parts. The first part is the logger which is attached to an enclave application before starting. Attaching is done using the `LD_PRELOAD` functionality of the dynamic linker to preload the logger before the SGX SDK. This allows the logger to hook into the SDK and intercept all ECalls and OCalls to instrument them. Using the logger in this way enables usage of sgx-perf without the need to change the enclave application in any way. The logger does not only capture OCalls created by enclave developers, but also captures the OCalls integrated into the SGX SDK itself which are used to implement synchronization primitives.

Since an enclave cannot wait for an interrupt, Intel built synchronization primitives into the SDK that will leave the enclave and then call the appropriate system and library calls outside the enclave. On first look, that seems like a bad design decision as, e.g, acquiring a lock should be a very fast operation but now seems to require an OCall which in turn requires an enclave transitions. However, the SDK implements a hybrid approach. Lock information is managed inside the enclave and acquiring a lock is therefore fast if the lock is not held. If the lock cannot be acquired, only then the enclave is left after marking the lock such that the current lock holder knows that another thread is waiting outside. When the current lock holder releases the lock, the thread will also perform an OCall to wake up the waiting thread. This optimization mechanism allows high performance when lock contention is low but creates a high overhead if lock contention is high due to OCalls.

All gathered data is logged into an SQLite database to make it easy for third-party tools to interpret and work with the data. sgx-perf also comes with its own analyser, the second main part, to interpret the data. The analyser identifies different performance problems based on the timing behaviour of the ECalls and OCalls and offers recommendations on how to change the code to eliminate these problems. The evaluation showed that sgx-perf is capable of making recommendations that increase enclave performance by up to 2.16$\times$ in four non-trivial SGX workloads.

While the main part of sgx-perf is gathering and analysing data on enclave transitions, it also is capable of measuring an enclave's working set. sgx-perf was published at a time when EPC sizes were fixed and small (up to 188 MiB) and enclave working set management was a valid concern. Today, with flexible EPC, these concerns are less relevant but still valid as software with big working sets such as DBMSs might fill up most of the hosts memory and therefore might cause EPC page swapping. sgx-perf can still help in these cases to figure out if an enclave is too big and to fine tune EPC size allocations on the host.

3.2 Related Work

Before the introduction of SGX, analysing application performance was possible using a multitude of language-agnostic and language-dependant tools, sometimes even with support from the operating system kernel.

For example, on Linux there is *perf* [12], a frontend for the performance counter subsystem of the kernel. Performance counters are hardware registers that collect information about, e.g., number of executed instructions, cache-misses or branch mispredictions. The subsystem can also collect information about system calls, socket usage and filesystem operations. Although the hardware performance counters are global, the subsystem can be instructed to filter only a specific task which enables application profiling. While perf is language-agnostic, it is capable of annotating source code with event counters to show hot-spots if the profiled binary contains debug symbols. Therefore, this only works for compiled languages and not interpreters or JIT compilers.

A similar tool is offered by Intel under the name *VTune Profiler* (formerly VTune Amplifier) [49]. It offers a wide range of possibilities, such as analysis of accelerator and GPU usage, hot-spot analysis including flame graphs, threading analysis and more while supporting multiple languages such as C, C++, C#, Java, Python and Go. Originally developed to find memory leaks, the *Valgrind* [65] toolkit nowadays also contains tools for profiling, namely the *Massif* heap profiler, the *Cachegrind* cache profiler and the *Callgrind* call graph analyser which work best with C and C++ applications.

Some programming languages, such as interpreted or JIT compiled languages, also have their own profiling capabilities and tooling. For Python, the *cProfile* [14] module offers function call and hot-spot profiling useable directly from Python code. cProfile is a popular choice for Python applications and is used by popular projects such as Home Assistant [13]. For Java, a wide range of tools exist such as Oracle's profiling and event collection framework *Java Flight Recorder* [68] or JProfiler [34].

When SGX was released, the need for performance analysis of SGX enclaves could only be fulfilled by VTune as Intel updated their profiler to work with SGX by offering hot-spot analysis for SGX enclaves. However, multiple papers quickly identified that SGX enclaves have some unique performance characteristics which developers need to be aware of. *SCONE* [9] and *SecureKeeper* [22] both measured the impact of SGX paging when an application exceeds the available EPC of a system. EPC paging is so costly, that it should be avoided completely, if possible. Similarly, Weisse et al. [98] and Zhao et al. [102] have both shown that enclave transitions are costly, and their numbers need to be kept low. Their proposals of asynchronous enclave transitions and a custom memory allocator have been explored in SCONE and *Eleos* [70], respectively. While those papers show the importance of knowledge of the new performance characteristics, they do not offer any way for developers to check how badly their applications are affected.

Building on those works, Gjerdrum et al. [33] presented a list of SGX performance principles and recommendations for enclave developers in a cloud scenario. The authors do not directly recommend minimizing enclave transitions, instead they focus on the size of the data that needs to be copied during an enclave transition, and recommend keeping the data size small. While it is certainly true that copying larger amounts of data takes longer, the transition itself is still costly, even if the data is kept small. They also advocate to keep enclave size small to avoid EPC paging and to increase start times. Although the paper contains recommendations for developers, it, again, does not provide any way for developers to check their applications.

sgx-perf fills this gap by offering profiling support for these newly identified performance characteristics and providing automatic recommendations to developers on how to optimize their application around those characteristics. Since the publishing of sgx-perf, other related systems have been developed shown below.

SGXoMeter [58] is a framework that allows developers to benchmark their code inside a controlled SGX environment. Its main purpose is to unveil performance differences between code running inside an enclave and code running outside an enclave. The architecture of SGXoMeter allows developers to only write their code once and then benchmark it both ways. In previous works[9, 22], the overhead of running inside an enclave was shown to be measurable but negligible when staying inside EPC limits. With SGXoMeter, these results were reproduced, however, it uncovered that different SGX SDK version can lead to different performance due to internal changes that, e.g., affect memory allocation, thread selection and side channel mitigations. Certainly, it could also be used to gauge the effect of µcode changes to real-world enclave performance. Compared to sgx-perf, SGXoMeter is not designed to benchmark an existing enclave, but instead offers a mechanism to benchmark specific functions and algorithms inside and outside an enclave to compare performance characteristics.

TEEMon [54] is a real-time performance monitor for SGX enclaves. It provides fine granular metrics of the running enclave to either a monitoring system for real-time analysis of an enclavized service or for a SGX framework to consume for real-time tuning of the enclave. TEEMon focuses on providing real-time metrics and, unlike sgx-perf, does not offer a way to get action recommendations based on the gathered metrics. Due to its architecture, it's meant to be used at runtime after enclave deployment and not during the development phase and while it can work with the SGX SDK, it is more tailored to a library OS enclave.

A similar focus is made by *SGXTuner* [59] which offers automatic tuning of the standard library embedded inside a library OS enclave. Compared to enclaves built with the SGX SDK, library OS enclaves come with a lot of subsystems that can be tuned, such as system call queues or user-level threading. SGXTuner also supports tuning application specific parameters, such as the size of buffers, timeouts or other limits.

(Publication starting next page)

sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves

Nico Weichbrodt
IBR, TU Braunschweig
Germany
weichbr@ibr.cs.tu-bs.de

Pierre-Louis Aublin
LSDS, Imperial College London
United Kingdom
p.aublin@imperial.ac.uk

Rüdiger Kapitza
IBR, TU Braunschweig
Germany
kapitza@ibr.cs.tu-bs.de

ABSTRACT

Novel trusted execution technologies such as Intel's Software Guard Extensions (SGX) are considered a cure to many security risks in clouds. This is achieved by offering trusted execution contexts, so called *enclaves*, that enable confidentiality and integrity protection of code and data even from privileged software and physical attacks. To utilise this new abstraction, Intel offers a dedicated Software Development Kit (SDK). While it is already used to build numerous applications, understanding the performance implications of SGX and the offered programming support is still in its infancy. This inevitably leads to time-consuming trial-and-error testing and poses the risk of poor performance.

To enable the development of well-performing SGX-based applications, this paper makes the following three contributions: First, it summarises identified performance critical factors of SGX. Second, it presents *sgx-perf*, a collection of tools for high-level dynamic performance analysis of SGX-based applications. In particular, *sgx-perf* performs not only fined-grained profiling of performance critical events in enclaves but also offers recommendations on how to improve enclave performance. Third, it demonstrates how we used *sgx-perf* in four non-trivial SGX workloads to increase their performance by up to 2.16x.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**;
• **Security and privacy** → *Software security engineering*;

KEYWORDS

Intel Software Guard Extensions, Trusted Execution, Performance Profiling

ACM Reference Format:
Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274808.3274824

1 INTRODUCTION

Although cloud computing has become an everyday commodity, customers still face the dilemma that they either have to trust the provider or need to refrain from offloading their workloads to the cloud. With the advent of Intel's Software Guard Extensions (SGX) [14, 28], the situation is about to change as this novel trusted execution technology enables confidentiality and integrity protection of code and data – even from privileged software and physical attacks. Accordingly, researchers from academia and industry alike recently published research works in rapid succession to secure applications in clouds [2, 5, 33], enable secure networking [9, 11, 34, 39] and fortify local applications [22, 23, 35].

Core to all these works is the use of SGX provided *enclaves*, which build small, isolated application compartments designed to handle sensitive data. Enclave memory is encrypted at all times and integrity checks by the CPU detect unauthorised modifications. Internally, enclaves are a special CPU mode and are enabled via new instructions. To ease development of enclaves, Intel released a Software Development Kit (SDK) [16]. It hides the SGX hardware details from the developer and introduces the concept of ecalls and ocalls for calls into and out of the enclave, respectively, that look like normal functions calls. While enclaves offer confidentiality of data and integrity of code and data, these properties come with a performance cost [1, 31, 44]. However, despite the rapid research progress over the last years, the understanding of the provided hardware abstractions and the offered programming support – especially its performance implications – is still limited. This leads to time consuming *trial-and-error* development and debugging as well as incurring the risk of bad performance.

Early works such as SCONE [1], SecureKeeper [5], and Eleos [31] have shown that enclaves have multiple potential performance issues that can be addressed through different techniques such as asynchronous calls [1, 44] and extended memory management support [31]. However, all these systems provide isolated solutions and only slightly address the development of commodity applications using the Intel SGX SDK. To support the SDK, Intel updated their low-level performance profiler *VTune Amplifier* [18] to allow profiling of SGX enclaves. However, VTune is built for performance profiling on an instruction level, providing information about hot spots in functions. While this is helpful, it does not provide information and insights about the specific characteristics of SGX. In summary, while SGX is rapidly adopted to secure applications, there is limited knowledge and a severe lack of tooling support empowering users to implement well-performing applications.

In this paper we aim to address this demand by a tripartite approach. First, §3 provides a summary of the performance critical factors of SGX. Second, §4 presents *sgx-perf*, a collection of tools to dynamically analyse enclaves, without having to recompile the application. *sgx-perf* allows developers to trace enclave execution and record performance critical events such as enclave transitions and paging. It does so by shadowing specific functions of the SGX

SDK and thereby redirecting the control flow. Analysing the recorded data then gives insights on potential bottlenecks. Furthermore, *sgx-perf* offers SGX-tailored recommendations on how to improve the enclave code and interface to increase performance. Third, in §5 we analyse enclaves of multiple projects using *sgx-perf*, implement recommendations when applicable to improve performance and present our findings. In particular, we looked at four classes of applications that are relevant for cloud enviroments and SGX: a cryptography library [2], a key-value store [5], an application partitioned using the Glamdring tool [25] and a database [37]. We found that the enclave interface design is an integral part of enclave performance and that applying the recommendations from *sgx-perf* increases performance by at most 2.66×.

In addition, §2 gives background information on Intel SGX, the SGX SDK, existing tooling support, and why enclave performance matters, §6 shows related work and §7 concludes.

2 BACKGROUND

This section gives an overview about SGX and the available programming support for enclaves as provided by the SGX SDK. Furthermore, we present enclave performance considerations and current SGX-aware profiling tools.

2.1 Intel Software Guard Extensions

Intel's Software Guard Extensions (SGX) [28] is an extension to the x86 architecture, which allows the creation of secure compartments called *enclaves*. Enclaves can host security critical code and data for applications running on untrusted machines. Authenticity and integrity of the enclave is guaranteed by SGX through both local and remote attestation mechanisms.

The memory used for enclaves is a special region of system memory, called the Enclave Page Cache (EPC). In current SGX capable systems it has a maximum size of 128 MiB, of which ≈93 MiB are usable. While enclaves can be bigger than this limit, this incurs costly swapping of pages to and from the EPC. All enclave memory is fully and transparently encrypted as well as integrity protected.

Inside the EPC, each enclave has its own page holding metadata about the enclave such as its size and signature to check for its integrity, called *measurement*. Furthermore, each enclave has at least one Thread Control Structure (TCS) page describing an entrypoint into the enclave. TCSs are used by threads to enter the enclave. The number of TCS determines the maximum number of threads that can execute inside the enclave concurrently. Each TCS also points to its own stack inside the enclave. Lastly, enclave heap, code, and data sections are also located inside the EPC.

Enclave creation must be handled in kernel-space, e.g., through a kernel module, whereas enclave interaction is restricted to userspace applications. Privileged code cannot enter enclaves and unprivileged code cannot create enclaves. Entering an enclave is done through the `EENTER` instruction which changes the execution context to inside the enclave. It can be left again with `EEXIT`.

Entering and leaving the enclave are *synchronous* operations, i.e., they are done explicitly. Furthermore, there exists a way to *asynchronously* leave the enclave. Whenever an interrupt, exception, fault or similar happens while the processor is executing inside the enclave, then the current context, i.e., the state of the registers, is

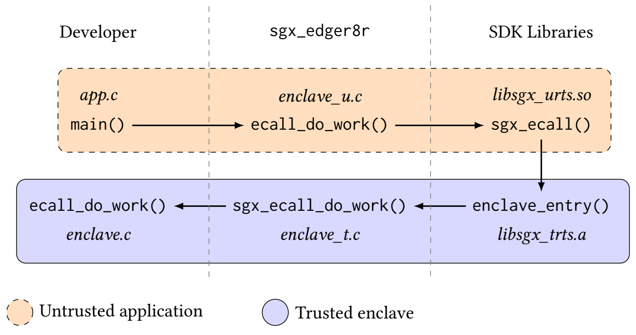


Figure 1: Architecture of an ecall. The developer provides the application and ecall implementations whereas the SDK generates code which uses the URTS and TRTS libraries.

saved into the thread-specific State Save Area (SSA). The current instruction is then finished and the enclave is left to handle the situation, e.g., call the interrupt handler. This is called an Asynchronous Enclave Exit (AEX). After the handler finishes, the processor executes the user-defined handler located at the Asynchronous Exit Pointer (AEP) instead of resuming the enclave. Typically, the handler uses the `ERESUME` instruction to continue enclave execution, which restores the saved context and continues at the point of interruption but re-entering with `EENTER` is also possible.

2.2 Intel SGX Software Development Kit

To ease enclave development, Intel released a Software Development Kit (SDK) [16] in 2016. The SDK abstracts the enclave transitions into a concept they call *enclave calls* and *outside calls*. Enclave calls, or ecalls, are calls from the untrusted application into the enclave. Outside calls, or ocalls, are calls in the opposite direction. Enclave developers specify the enclave interface in form of ecalls and ocalls using the Enclave Description Language (EDL). The SDK source-to-source code generator `sgx_edger8r` then generates wrapper code from this EDL file to be compiled and linked into the developed application and enclave. Furthermore, the SDK provides a trusted, but stripped down standard C/C++ library, a trusted cryptography library and Trusted Runtime System (TRTS) for the enclave as well as an Untrusted Runtime System (URTS) for the untrusted application. The cryptography library provides basic encryption and decryption functions whereas the TRTS and URTS handle the enclave transitions and call dispatching. Missing features from the standard C/C++ library that require system calls have to be reimplemented, e.g., as ocalls.

As can be seen in Figure 1, the actual enclave transitions are located in the URTS (`EENTER` and `ERESUME`) and TRTS (`EEXIT`). The SDK uses the same generic entry point for all ecalls with a trampoline dispatching the call to the right function. Similarly, ocalls are handled the other way round.

2.3 Enclave Performance Considerations

Enclave performance has been the subject of research since the availability of SGX-capable hardware in 2015. The consensus is that both enclave transitions and enclave paging are expensive and should be avoided. Several research projects propose different techniques to eliminate transitions and make better use of the memory consumption [1, 31, 44]. Unfortunately these require a change in programming paradigms and are not openly available like the SGX SDK.

2.3.1 Enclave Transitions. Enclave transitions are the base mechanism to be able to execute code inside the trusted execution environment. Furthermore, enclaves are restricted to a subset of the instructions available on the processor. In particular they cannot use `int` or `syscall` [14] and therefore cannot issue system calls, for I/O operations or threads synchronisation. These features thus require the implementation of additional ocalls.

Weisse et al. [44] measured enclave transitions of SDK ecalls and ocalls in the order of 8,600 to 14,000 cycles, depending on cache hit or miss. Instead, we directly measured the time elapsed between the `EENTER` and `EEXIT` instructions, excluding the overhead of the URTS looking for a free TCS and the TRTS actually dispatching the call, in three different settings: (i) on an unmodified Intel SGX-capable processor; (ii) after applying the SDK and microcode updates to fix the Spectre [20] speculative execution vulnerability, which also affects SGX [6, 29]; and (iii) after applying the microcode update to fix the Foreshadow (L1 Terminal Fault) [42] attack.

In the first case, we measured transition times of $\approx 5,850$ cycles ($\approx 2,130$ *ns*) with a warm cache for one round-trip (see §5 for the experimental settings). In the second case, we measured a transition time of $\approx 10,170$ cycles ($\approx 3,850$ *ns*), $\approx 1.74\times$ more than without patches. Finally, with all the updates and microcodes to address the Spectre and Foreshadow vulnerabilities enclave transitions became even slower, resulting in a round-trip time of $\approx 13,100$ cycles ($\approx 4,890$ *ns*), $\approx 2.24\times$ more. This further underlines the need to save on enclave transitions.

2.3.2 In-Enclave Synchronisation. Enclaves can be multi-threaded and therefore need synchronisation primitives. Unfortunately, as sleeping is not possible inside enclaves, the in-enclave synchronisation primitives provided by the SGX SDK implement additional ocalls to sleep outside of the enclave.

The SDK offers mutexes that work as follows: if a thread tries to lock an unlocked mutex, then this operation succeeds without needing to leave the enclave. Whenever a thread tries to lock an already locked mutex, it will put itself into a queue and exit the enclave via an ocall to sleep. The thread holding the mutex will then need to wake up the sleeping thread by looking into the queue and leaving the enclave via an ocall. A mutex lock can therefore result in two ocalls. This is especially a problem as the wake-up ocall is typically very short (<10µs) and therefore the enclave transition is taking the majority of the time.

2.3.3 Enclave Paging. Another important factor for enclave performance is enclave size, especially the size of the working set. SGX stores all enclaves inside the EPC which on current implementations has a size of 128 MiB. Of those, 93 MiB are usable; the difference is used to store metadata used for integrity protection [10].

In the EPC, enclaves basically consist of four parts: one metadata page, its code, the heap and a thread-data page (TCS), stack and SSA pages for each configured enclave thread. The heap and stack sizes are set at enclave build time via a configuration file and should be large enough to accommodate all needed dynamic memory allocations. Contrary to normal application development, the heap and stack are not *virtually infinite*, but actually have a limit that can be hit if developers are not cautious. Therefore, one might be tempted to increase their sizes, or even the number of maximum concurrent threads, to some high number. With SGX v2, this becomes less of a problem, as the enclave can be extended after creation. Therefore, the enclave can be created small and as soon as stack or heap are exhausted, new pages may be added on-demand. Clearly, this still incurs paging if the enclave exceeds the EPC size.

SGX supports paging from EPC to main memory to accommodate enclaves that do not fit into EPC. However, these operations are costly and have a big impact on enclave performance [1, 5]. This is due to the cost of added enclave transitions to handle page faults as well as extra computation needed for cryptographic operations. Therefore, carelessly increasing and using enclave memory might incur paging and therefore performance hits.

2.4 Existing Tooling Support

Since SGX essentially adds a new processing mode, most existing tools inspecting processes do not expect enclaves and, therefore, are not able to interact with them. To our knowledge, only the following two tools support SGX in some way.

The SDK ships with a plugin for the GNU Debugger (gdb), allowing it to inspect enclaves[1], set breakpoints and more. The separate application and enclave stacks are virtually stitched together to display a single call-stack for calls inside the enclave to ease debugging. This plugin only works for applications developed with the SDK, other projects like SGX-LKL [27] also support gdb with their own plugin.

Intel updated their profiling software *VTune Amplifier* [18] to work with SGX. VTune is able to do a so-called *sgx-hotspots analysis* on applications utilizing enclaves which gives developers insight into their enclave functions regarding execution hotspots. A hotspot is a piece of code that is executed frequently, e.g., the body of a loop, defined by metrics such as overall cycles per instruction or cache misses. Knowing where hotspots are can help developers to decide which code parts to optimise further. VTune focuses on low-level analysis of code fragments only.

Unfortunately, these tools are not sufficient to help the developer write efficient enclave code as they do not take into account SGX specific features.

3 SGX PROBLEMS AND SOLUTIONS

As outlined in §2.4, the metrics collected by current tools are not sufficient to tackle the performance problems of enclaves. According to previous research projects [1, 31, 44], the overhead of using enclaves primarily boils down to (i) the number of enclave transitions during execution and their duration; and (ii) the number of paging events.

Paging events perform SGX-specific computations while also causing enclave transitions due to fault handling. Therefore, reducing the number of enclave transitions should be prioritised. This can be achieved through a well-designed enclave interface that both maximises the execution time spent either inside or outside the

[1]This only works on enclaves that have the *debug* flag set.

Problem	Solution
Short Identical Successive Calls	Batch calls Move caller in/out encl.
Short Different Successive Calls	Merge calls Move caller in/out encl.
Short Nested Calls	Reorder calls Duplicate ocalls
Short Synchronisation Calls	Lock-free data structures Hybrid sync. primitives
Paging	Reduce memory usage Load pages before ecall Do not use SGX paging
Permissive Enclave Interface	Limit public ecalls Limit ecalls from ocalls Check data and pointers

Table 1: Identified performance and security problems and their possible solutions.

enclave and minimises the number of transitions during execution. This leads us to our premise that calls whose raw execution time is shorter than the enclave transition time should be avoided if at all possible. In addition, we argue that the robustness of the enclave interface is of prime importance and that it is necessary to analyse it to look for potential security problems.

The rest of this section details SGX-specific problems that can arise in practice regarding the performance and security of enclaves as well as recommendations to improve the code. A summary can be found in Table 1.

3.1 Short Identical Successive Calls

The Short Identical Successive Calls (SISC) problem occurs when multiple short executions of the same call are made in succession. As transitions have a fixed cost, computations that are shorter than it are wasteful. Therefore, multiple calls of the same ecall entering or multiple calls of the same ocall leaving the enclave in succession should be ***batched***.

Another solution can be to ***move the caller function inside/outside of the enclave***. As a result, only one transition will occur for the successive calls. See §5.2.3 for an example. Note that moving a function from inside the enclave to outside, to remove successive ecalls, might pose a security risk as the ecall probably handles sensitive data. A security evaluation is therefore recommended when moving functions outside of the enclave.

3.2 Short Different Successive Calls

Contrarily to SISC, a Short Different Successive Calls (SDSC) problem occurs when multiple short executions of *different* calls are made in succession. Same as with SISC, this causes a waste of resources as actual computation time might be less than transition time. Possible solutions are ***merging*** these calls into a single call or ***moving the caller function inside/outside the enclave***. See §5.2.2 for an example.

3.3 Short Nested Calls

The Short Nested Calls (SNC) problem occurs when short calls are made at start or end of another call. These short calls are candidates for possible elimination as their execution should either be done before or after the call instead of during the call. However, this might not always be possible due to the application's architecture. An example for this is an ecall that issues an ocall to allocate memory for a result. Instead of allocating this memory during the ecall, the allocation should be moved to before the ecall. The solution is therefore to ***reorder*** the ocall to execute before the ecall.

This might be problematic if the needed space is not known before the ecall's execution. However, in this case a sensible default can be chosen and an ocall can be issued only if more memory is needed. Both SecureKeeper [5] and LibSEAL [3] use similar techniques to circumvent issuing ocalls for untrusted memory allocations during ecalls.

The solution is not exclusive to ocalls during ecalls, it can also be applied to short ecalls during ocalls. Depending on the call, short ocalls can also be ***duplicated*** inside the enclave. This increases the Trusted Code Base (TCB) of the enclave but also improves performance.

3.4 Short Synchronisation Calls

A special case of SNC are Short Synchronisation Calls (SSC). As stated in §2.3.2, the SDK provides in-enclave synchronisation primitives that potentially issue ocalls for sleeping and waking up threads. The wake-up ocalls are typically very short (<10µs on average in all cases we observed) whereas the sleep calls can vary in execution time, depending on how long the thread is sleeping. Short sleep calls suggest that the time the lock is taken is very short and going outside of the enclave for sleeping should be avoided.

In these cases, it would be beneficial to have a hybrid locking mechanism that first tries to take the lock inside the enclave multiple times in a spinlock fashion before going to sleep or, if possible, to use non-blocking data structures.

3.5 Paging

As stated in §2.3.3, paging events during enclave execution are very costly due to additional transitions and cryptographic operations. Enclaves too large for the EPC can be the result of having a too large dataset inside the enclave or of poor data handling inside the enclave. Developers need to be aware that the need for space-efficient data structures is higher for enclaves than other applications.

Paging can be mitigated by multiple techniques: (i) keep the enclave small to always fit into EPC, (ii) prevent page faults during enclave execution by pre-loading pages into the EPC or (iii) use an alternative memory management mechanism inside the enclave instead of the SGX paging mechanism. (i) can be achieved by using space-efficient data structures or by loading smaller chunks of data into the enclave, if possible. However, this might not be enough as the EPC is shared between all running enclaves. It is not possible to assume which enclave size is suitable as the EPC might already be blocked by other enclaves and paging is unavoidable, especially in a multi-tenant cloud scenario. (ii) is possible by loading the needed pages before issuing the ecall. This prevents the costly page faults and AEXs inside the enclave during execution. Examples of (iii) have been implemented by the Eleos [31] and STANlite [33] systems. In a nutshell, these systems store sensitive data in an encrypted and integrity-protected manner outside of the enclave, in the untrusted environment. Then, when the data is needed, it is copied inside the enclave and decrypted.

In general, enclaves should be designed to encounter paging as seldom as possible as it incurs too high performance costs through additional transitions.

3.6 Security Enhancements

Given that enclaves deal with sensitive data inside an untrusted environment, it is necessary to reduce the attack surface of their interface [17]. We have observed three possible security problems that can easily be mitigated.

First, the SGX SDK allows ecalls to be defined as *public* or *private* [15]. Public ecalls can always be called whereas private ecalls can only be called during an ocall. Defining an ecall as private can enhance the enclave security by limiting the possible paths leading to an ecall. It is then easier for developers to make assumptions about the state the enclave is in when executing a given ecall.

Second, the developer has to precisely specify which ecalls are allowed within each ocall. If a particular ecall has been forgotten, an error will be triggered during execution. Developers might be tempted to simply allow every ecall from all the ocalls. In the worst case, if a specific ecall/ocall combination is not considered by the developer, this could be exploited by an attacker to change the control path of the execution of the program and gain access to enclave secrets. Consequently, it is important to limit the ecalls that can be called from any ocall.

Third, the EDL file defines the behaviour of pointers passed as arguments of the ecalls and ocalls: *in*, if data has to be copied inside (resp. outside) the enclave before an ecall (resp. ocall); *out*, if data has to be copied outside (resp. inside) the enclave after an ecall (resp. ocall); and *user_check*, if handling the pointer is left to the developer. While *user_check* is the simplest behaviour, it might also lead to security vulnerabilities, e.g., due to buffer overflows, time-of-check-to-time-of-use attacks [43] or passing an in-enclave address [19]. It is thus important to check and limit how the pointers are passed and used across the enclave interface.

4 THE *SGX-PERF* TOOLS

In this section we present *sgx-perf*, a toolset to analyse performance-impacting behaviour of enclaves. It pinpoints the problems mentioned in §3 and gives developers hints on how to restructure their enclaves to avoid these issues.

sgx-perf consists of multiple tools that work together: an event logger, the working set estimator and an analyser. Event recording is done by the event logger which traces ecalls, ocalls, AEXs and EPC paging. Working set estimation is done by a separate tool, as it heavily interferes with enclave execution. Lastly, analysis and visualization of the data is done by the analyser.

The *sgx-perf* event logger is implemented as a shared library. This library is preloaded into the untrusted application using the `LD_PRELOAD` environment variable so the dynamic linker loads it before all others including the URTS. This makes it possible to use the event logger without having to modify the untrusted application,

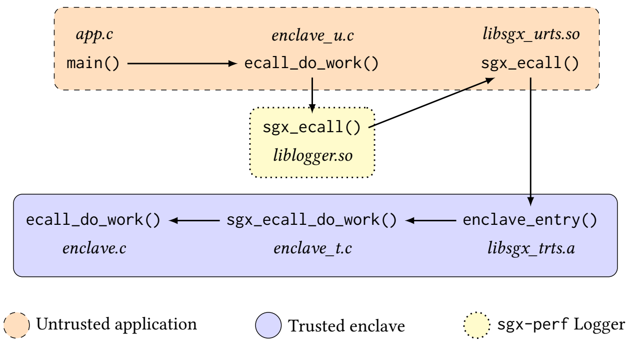


Figure 2: *sgx-perf* tracks ecalls by shadowing the call to `sgx_ecall` so it is called instead of the URTS.

the enclave or the SDK. Function calls are traced by providing the traced symbols anew. For example, the logger provides its own implementation of `pthread_create` which is then called by the application instead of the real function inside the standard library. It can trace the call and record an event before dispatching the call to the real implementation.

Additionally, the logger registers its own signal handlers for some signals. The handler registering functions `signal` and `sigaction` are also overloaded, so that other registered handlers can be saved and called after the logger has processed the signal itself. This is important for tracing some applications, e.g., Java applications with enclaves attached via Java Native Interface (JNI), as the OpenJDK uses signals for communication between threads.

All events are serialised to a SQLite database. This makes it possible to analyse the data with other tools without having to implement parsing of the data. Migrating the data to a real SQL server can also be envisioned.

4.1 Tracing ecalls and ocalls

The main method of interaction with enclaves are ecalls and ocalls which cause enclave transitions. As described in §2.3.1, we know that enclave transitions are costly and if high performance is desired, their count needs to be minimised. Furthermore, short calls into or out of the enclave are also not desirable as the overhead of transitioning can overshadow the actual computation time.

To show the ecall and ocall behaviour of an application, the logger traces these transitions as described in the following.

4.1.1 Tracing of ecalls. To use ecalls, the application developer has to describe the enclave interface and generate wrapper code. This wrapper code allows the developer to call the ecall functions by their given name (e.g., `ecall_encrypt`) like a normal function. In practice, the symbols exists twice, once inside the enclave and once outside. The outside wrapper calls the `sgx_ecall` function of the URTS with a generated numeric identifier which causes an enclave transition into a trampoline that resolves the identifier to the actual ecall and calls it.

This design of issuing all ecalls through a common function inside the URTS allows the logger to shadow the implementation of `sgx_ecall` with its own to trace calls into the enclave (see Figure 2). When the `sgx_ecall` function of the logger is called, it first records

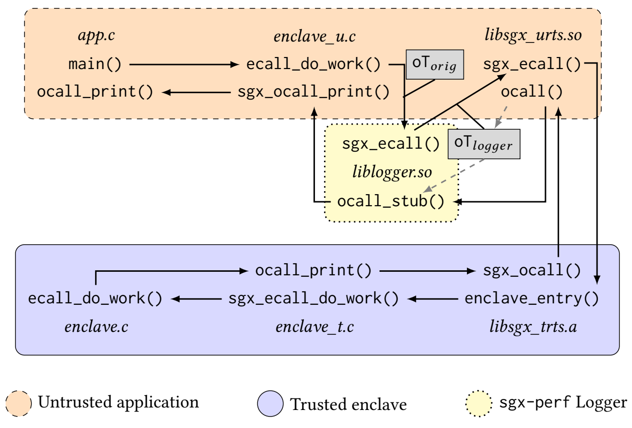


Figure 3: *sgx-perf* rewrites the ocall table $\mathtt{oT}_{orig}$ to its own table $\mathtt{oT}_{logger}$ during ecalls to track ocalls.

the current time as well as the identifier of the issuing thread and the ecall identifier. It then calls the `sgx_ecall` function of the URTS. Finally, it again records the current time in order to measure the duration of the ecall. Note that the logger is executing outside of the enclave and is therefore able to measure time.

4.1.2 Tracing of ocalls. To trace ocalls we tried to employ the same mechanism as for ecalls as the design for calling ocalls is basically the same: a common `sgx_ocall` function dispatches the call based on an identifier. Unfortunately, this function is part of the TRTS and therefore inside the enclave. The logger cannot shadow an enclave function as this would violate the enclave's integrity.

The `sgx_ocall` function uses the `EEXIT` instruction to leave the enclave which needs the address of the ocall function to jump to. These addresses are not fixed, as the ocalls could be inside shared libraries or because the binary is relocated by the Operating System (OS). This makes it impossible for the SDK to include the addresses into the enclave during compilation, therefore they have to be injected at runtime.

The SDK chooses the following approach: It constructs a table mapping numeric identifiers to function pointers called `ocall_table` which is given as an argument to `sgx_ecall`. The pointer to the table is then saved inside the URTS for later use. Should an enclave issue an ocall, it will exit the enclave to a function that will look up the function pointer from the saved ocall table. This makes it possible for the logger to change the table and inject our own. However, the function pointers included in the original table are already pointing to the correct ocall functions and not to a common function, e.g., a trampoline, that we could intercept.

Therefore, as seen in Figure 3, a call stub is generated by the logger on the fly for each function in the table. The call stub is given information about the ocall like its identifier, the enclave identifier and the original function pointer. Then, when an ocall happens, the generated call stub is called instead, which logs the appropriate events and then calls the original ocall. All stubs are combined as a new table (oT_{logger}) which is propagated in place of the original one during the ecall tracing. This means, that we always replace the table, even if the ecall does not perform any ocalls, as we cannot know this beforehand.

Call stub and table creation is only needed once per ocall table. In practice, this means exactly once per enclave as SDK applications have one ocall table per enclave. Note that timestamps recorded do not include transition times as they are recorded outside of the enclave. This results in ocalls being seemingly shorter than ecalls when doing the same work as ecall timestamps include the transition time. For the analysis phase this means that for ocalls the execution time can be compared directly to the transition time whereas for ecalls, the transition time has to be subtracted from the measured execution time first.

4.1.3 Tracing In-Enclave Synchronisation. As stated in §2.3.2, the SDK supports special in-enclave synchronisation primitives that use ocalls to put threads to sleep. Through its ocall tracking facility, the logger can track these ocalls in a general way. In addition, the logger overloads the four specific synchronisation ocalls of the SDK: (i) sleep, (ii) wake up one, (iii) wake up multiple and (iv) wake up one and sleep. These four ocalls can be reduced to two event types: sleep and wake-up. The events allow the logger to also track which thread wakes up which other threads to track dependencies between them. This information can be used to detect high-contention scenarios that cause a high frequency of ocalls.

4.1.4 AEX Counting and Tracing. While executing inside an enclave, interrupts and faults can still occur. These need to be handled by the untrusted operating system and therefore the enclave has to be exited. For this, the concept of an AEX exists which saves the enclave state and then leaves the enclave to execute, e.g., the interrupt handler. Afterwards, a jump to the address pointed to by the AEP is made, which then decides whether to resume the enclave or do something else (see §2.1).

In the SDK, the AEP points to exactly one instruction, namely `ERESUME` which resumes the enclave. The logger can optionally patch this location with a jump to its own AEP. This allows it to either only count the number of AEXs per ecall or to record also the time at which each AEX occurred. This information is useful in conjunction with ecall duration, as longer ecalls are subject to more AEXs. Similarly, AEXs increase ecall duration as they interrupt them. Tracing AEXs allows the analyser to correlate ecall duration with AEX times as multiple AEX in short succession will delay an ecall significantly while not being an issue with the ecall itself. Such bursts of interruption can be caused by high system load or other external factors. For example, a high amount of interrupts on the core currently processing the enclave will result in an high amount of AEXs. Knowledge of this is helpful to separate high-interrupt execution, e.g., a network thread, from enclave execution by pinning the threads to different cores.

Due to a limitation in the first version of SGX, it is not possible to infer the reason for the AEX. While we can distinguish interrupts from some type of faults (e.g., segmentation faults, as those will engage a signal handler), we cannot differentiate interrupts from simple page faults. SGX v2 will enable this, as the SGX subsystem can be instructed to record the exit type into the enclave state. This type could then be read by the logger as long as the enclave is a debug enclave to further give the reason for the enclave exit.

However, even though the AEX cause is not recorded, the logger can still determine paging events, as shown in § 4.1.5.

4.1.5 EPC Page Tracing. Another problem with SGX enclaves is the limited space for the EPC. The EPC holds all enclave pages and is limited to 93 MiB. If the EPC is full, the SGX driver swaps pages to untrusted memory. This requires re-encryption of the page and incurs a heavy performance overhead as previous research has shown [1]. Ideally, enclave pages should never leave the EPC when the enclave is in use.

As paging happens inside the kernel, it is only possible to track it using kernel tracing approaches. The logger uses `kprobe` [21] to trace the respective functions inside the kernel driver that page in and page out enclave pages. This allows recording not only the time at which the swap happened, but also the virtual address of the page. Referencing those with the known enclaves of the process allows the logger to find out when and which part of an enclave has left the EPC. This information can be used to, e.g., determine enclave parts that were never actually used.

4.2 Enclave Working Set Estimation

In §3.5, we claimed that enclaves should be designed to seldom encounter paging. As this is potentially hard to achieve, *sgx-perf* comes with a tool that enables developers to get information about the working set of their enclaves on a page granularity, which is useful for right-sizing enclaves.

The working set is a metric that cannot directly be inferred from the size of the enclave binary. Enclaves do contain pages that can be safely paged out, as they are normally never used. These pages are either guard pages, e.g., for the enclave stack, or padding pages which are normally not accessed, but are needed as they are contained in the enclave measurement and the enclave size needs to be a power of two bytes.

The working set of pages is therefore much smaller than the actual enclave. To figure out the working set, *sgx-perf* provides a tool that tracks all accessed pages: the working set estimator. It reports the amount of pages accessed between two configurable points in time and operates by stripping all page permissions from enclave pages, catching access faults and restoring permissions on access. This works due to the fact that page permissions are saved and checked twice, once by the Memory Management Unit (MMU) and once by SGX. While the SGX permissions are fixed after enclave creation time[2], it is possible to modify the MMU page permissions during runtime, which are checked first. Missing permissions therefore lead to access faults when pages are accessed. Catching the faults and restoring permissions allows the working set estimator to track page accesses and determine the working set. This method is similar to the page tracing done by some SGX attack papers [43, 45]. In these cases, the page tracing is used to determine control flow of the enclave whereas in our case we just count the accesses. A page-table based approach, i.e. looking and clearing the access bits, would also work but requires kernel involvement which we wanted to avoid.

However, this approach has the disadvantage that we only see pages that are accessed during execution. We can't infer all possible branches taken during execution and therefore have to rely on different enclave inputs to give us an exhaustive list of page accesses. Figuring out which pages are accessed or not can only be done via exhaustive execution.

[2]Changing these is possible from inside the enclave with SGX version two. Software support is already available in the SGX SDK since v2.0.

4.3 Data Analysis and Developer Hints

The main objective of *sgx-perf* is to give developers information about their application's performance as well as hints on how to improve it. This is achieved using the analyser. In the following sections, we describe what information is provided by the analyser, which criteria are used to detect problems and what hints are given in these cases.

4.3.1 General Statistics. To give a first overview of the application, the analyser will calculate general statistics for all ecalls and ocalls. These statistics comprise number of calls, average and median duration, standard deviation as well as 90th, 95th and 99th percentile values. Furthermore, the analyser can generate histograms for the call execution times as well as scatter plots showing the call's execution times over the course of the application's execution. This information gives a quick overview over the calls and can be used to detect outliers. The analyser can also generate call graphs detailing dependencies between ecalls and ocalls to get an overview of the application's call patterns (see Figure 5 in §5.2.1).

4.3.2 Problem Detection. The main goal of the analyser is to give hints to developers regarding changes that can impact performance positively. In §3 we already detailed which performance problems can exist and how to mitigate them: Short Identical Successive Calls (SISC), Short Different Successive Calls (SDSC), Short Nested Calls (SNC), Short Synchronisation Calls (SSC) and paging. The analyser finds these issues and offers possible mitigation strategies such as ***batching*** or ***reordering***, ***merging***, ***moving*** or ***duplicating***, as shown in §3. For all five mitigation strategies the analyser tries to find opportunities to use them by analysing the calls made by the application. The overall intuition is, that a call experiencing many short executions needs to be optimised more than one experiencing only few. Therefore, the analyser mainly works by weighting ratios of call execution times. As a transition into the enclave and back out again takes ≈ 5µs on a fully patched system, we chose to look at calls with execution times below 10µs. Furthermore, the analyser tries to narrow the enclave interface, e.g., by finding ecalls that can be made private. It is the responsibility of the developer to check the applicability of the given recommendations. *sgx-perf* does not know about the internals of the applications and therefore cannot know if some recommendations cannot be applied due to design or application logic constraints.

Direct and Indirect Parents. For all analyses it is necessary to know which call has been issued before the call that is currently looked at. For ocalls during ecalls and ecalls during ocalls we have a simple relationship that is logged by default and called *direct parents*: An ecall E is a direct parent of an ocall O if and only if O was called during execution of E. The same is true for ecalls during ocalls. Contrary to direct parents, *indirect parents* are calls of the same type that were executed before the current call while belonging to the same direct parent.

$$\begin{tikzcd}
(1) & E_1 & E_2 \arrow[l, dotted] & E_3 \arrow[l, dotted] & (2) & E_1 & O_2^{E_1} & O_3^{E_1} \arrow[l, dotted] \\
(3) & E_1 & O_2^{E_1} & E_3^{O_2} & (4) & E_1 & O_2^{E_1} & E_3 \arrow[ll, dotted, bend right]
\end{tikzcd}$$

Figure 4: Some example for calls (C) with their direct (C^P) and indirect ($P \leftarrow C$) parents (P).

Figure 4 shows some calls and their indirect parents. Each E and O is an ecall and ocall respectively with their subscript numbers denoting their order with regards to time. Direct parents are denoted in superscript and indirect parents are referenced as a dotted arrow. As seen in (1), each ecall on the same level has the previous call as its indirect parent except for the very first ecall. This is the case when ecalls are called one after another. In (2) we see that only the ocall O_3 has O_2 as its indirect parent as they are both issued by E_1 and in (3) no calls have indirect parents. (4) shows a case in which the indirect parent of E_3 is not the previous call but rather the call before that one as O_2 is not of the same type as E_3.

Enclave Interface Security. The analyser is able of providing developers with hints regarding the security of the enclave interface. First, direct parents can be used to detect whether an ecall can be made private. If all instances of an ecall have direct parents, i.e., were issued during ocalls, then the analyser can recommend to make this ecall private and give a list of ocalls that need to be allowed to call it. Note that this recommendation is dependent on the workload.

Optionally, the analyser can be supplied the EDL file of the enclave. If so, it compares the current allowed ecalls for each ocall with those actually called. If they don't match, the analyser will show which ecalls should be removed from the set of allowed ecalls. The analyser will state the smallest set of allowed ecalls if no EDL is provided.

Furthermore, the analyser highlights calls which have pointer arguments annotated with `user_check` so that developers are reminded to look at these calls in particular whether all checks regarding the pointers are made.

Duplication and Moving Opportunities. Moving calls into or out of the enclave is a solution to the SISC and SDSC problems. Duplication of ocall functionality inside the enclave is a solution to the SNC problem. Detecting opportunities to apply the solutions is done by looking at the mean call execution times. Shorter execution times imply a stronger need for optimisation because more transitions can be saved. However, the ratio of short calls vs the total number of calls is also important: Only if the majority of executions are short, then the optimisation should be recommended. Thus, we arrive at Equation 1 with C_n stating how many calls were shorter than n µs and C_Σ being the total call count.

$$\left(\frac{C_1}{C_\Sigma} \geq \alpha\right) \vee \left(\frac{C_5}{C_\Sigma} \geq \beta\right) \vee \left(\frac{C_{10}}{C_\Sigma} \geq \gamma\right) \quad (1)$$

α, β and γ are configurable weights and default to $\alpha = 0.35$, $\beta = 0.50$ and $\gamma = 0.65$. These and the following weight values have been obtained through experimentation. In essence, the analyser checks if (i) 35% of calls (α) are shorter than 1 µs, (ii) 50% of calls (β) are shorter than 5 µs or (iii) 65% of calls (γ) are shorter than 10 µs. If the expression is true, a hint that this call should be moved across the enclave boundary to save transitions is displayed.

Reordering Opportunities. Call reordering is a solution to the SNC problem that is applicable to ecalls and ocalls. To detect reordering opportunities we check if calls are made after the start or before the end of another call. The analyser sets this in relation to the overall call count (C_Σ) as well as distance from the start/end by counting how many calls were made in the first (C^s) and last (C^e) 10 µs (C_{10}) and 20 µs (C_{20}) of a call. Equation 2 shows this for reordering opportunities at the start of calls. It is the same for reordering opportunities at the end of calls with C^s switched to C^e.

$$\left(\frac{C^s_{10}}{C_\Sigma} \times \alpha + \frac{C^s_{20}}{C_\Sigma} \times \beta\right) \geq \gamma \quad (2)$$

Again, α, β and γ are configurable weights and default to $\alpha = 1.00$, $\beta = 0.75$ and $\gamma = 0.50$. In essence, the analyser checks if the weighted calls (calls nearer to the start/end weigh more) are above the threshold γ. The call is flagged for possible reordering if the condition is true.

Merging and Batching Opportunities. For the SISC and SDSC problems batching and merging calls are the respective solutions. To merge or batch calls, the analyser cannot simply look at call frequency and execution time. Instead, it finds the *indirect parents* of each call and looks at the time difference between each indirect parent's end and the current call's start. Batching is a special case of merging and is applicable when the call is being its own indirect parent. Whether multiple different calls are flagged as mergeable into one is depicted by the expressions in Equation 3.

$$\frac{P_\Sigma}{C_\Sigma} \geq \lambda \wedge \left(\frac{P_1}{P_\Sigma} \times \alpha + \frac{P_5}{P_\Sigma} \times \beta + \frac{P_{10}}{P_\Sigma} \times \gamma + \frac{P_{20}}{P_\Sigma} \times \delta\right) \geq \epsilon \quad (3)$$

As before, α, β, γ, δ, ϵ and λ are configurable weights and default to $\alpha = 1.00$, $\beta = 0.75$, $\gamma = 0.50$ and $\delta = \epsilon = \lambda = 0.35$. First, the analyser only considers calls for merging, that are indirect parents at least 35% of the time (λ). P_Σ is the total call count of the indirect parent whereas C_Σ is the total call count of the current call. Then, the analyser checks how many indirect parents were 1, 5, 10 µs and 20 µs away, weights them accordingly (α, β, γ, δ, faster calls weigh more) and checks if the results is higher then the threshold ϵ. The call and indirect parent are flagged for possible merging/batching if the condition is true.

Recommendation Priorities and Security Implications. While all recommendations achieve the same results, i.e., less transitions, they do so in different ways. The analyser can recommend more than one optimisation per call. It is then up to the developer to decide which route to take with the following in mind: moving and duplication can increase the TCB of an application while reordering does not. Therefore reordering should be evaluated first before moving on to other recommendations. Furthermore, moving code out of the enclave should not be made without a security evaluation to avoid leaking enclave secrets. Contrarily, moving code into the enclave does not pose any additional security risk.

5 EVALUATION

Our evaluation answers the following questions: (i) what is the overhead of running an application with *sgx-perf*? And (ii) can *sgx-perf*

	(1) Single ecall	(2) ecall + ocall
Native	4,205 ns	8,013 ns
with Logging	5,572 ns	10,699 ns
Overhead	≈1,366 ns	≈2,686 ns
ocall only	–	≈1,320 ns
(3) Long ecall	Execution time	AEX count
with Logging	45,377 μs	–
AEX counting	45,390 μs	11.51
AEX tracing	45,390 μs	11.56
Overhead	per call	per AEX
AEX couting	≈14,612 ns	≈1,076 ns
AEX tracing	≈15,151 ns	≈1,118 ns

Table 2: Mean execution times per call and overhead of the logger overhead experiments. Variance is omitted as it is not significant.

detect optimisation opportunities in systems that use Intel SGX? To this end we evaluate *sgx-perf* with several microbenchmarks as well as four different applications: (i) TaLoS [2], a cryptography library, (ii) SecureKeeper [5], a key-value store, (iii) SQLite [37], a database and (iv) LibreSSL [30] partitioned with Glamdring [25]. Our evaluation first shows that the overhead of the event trace logging of *sgx-perf* is a fixed 1366 ns per call (see 5.1). Then, it shows that *sgx-perf* recommendations are useful to the developer as we were able to improve the performance by 1.33× to 2.66× after following them (see 5.2.3).

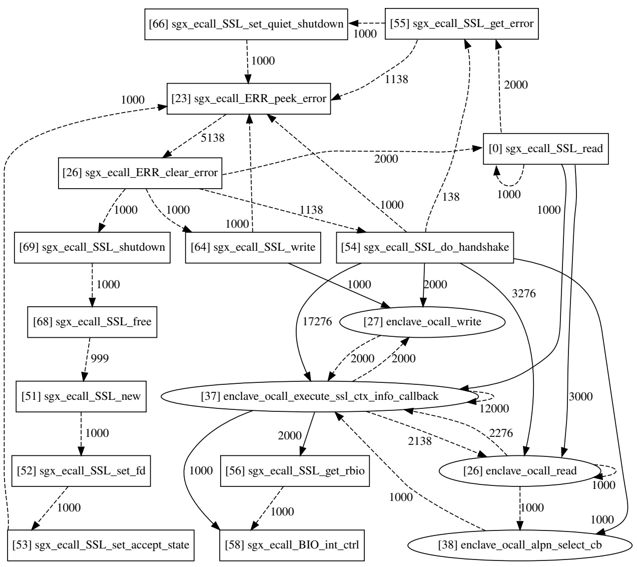
Experimental Settings. All the experiments were conducted on a system consisting of a Intel Xeon E3-1230 v5 @ 3.40 GHz processor, 32 GB (2×16 GB @ 1600 MHz) of memory and a 256 GB SATA-III SSD. We used Ubuntu 16.04.4 with Linux 4.4.0-116 with Kernel Page Table Isolation which mitigates the Meltdown [26] attack. If an application needs clients processes, they are executed on identical machines connected via a 10 Gbit/s ethernet link.

5.1 Performance Overhead of Logging

To measure the overhead of the event logger, we conducted three experiments: (1) a single ecall is executed n times; (2) a single ecall is executed n times. This ecall also performs a single ocall; and (3) a single ecall is executed n times. This ecall itself is executing a loop for k iterations doing nothing. For this experiment we also (i) counted or (ii) traced AEXs.

Each experiment has been executed 1000 times. For the experiments (1) and (2) we choose $n = 1,000,000$, for experiment (3) we choose $n = 1000$ and $k = 1,000,000$. For each run a warmup of 1,000,000 calls for (1) and (2), and 1000 calls for (3) respectively, has been used.

The results can be found in Table 2. As seen, the event logger adds an overhead of ≈1,366 ns per ecall. A similar result of ≈1,320 ns can be seen for ocalls. To find out the overhead of AEX counting and tracing, we performed experiment (3). In this experiment, a long running ecall is issued that will experience AEXs due to the timer interrupt. In this case, we made three measurements: attached logger without AEX counting or tracing, attached logger with AEX counting and attached logger with AEX tracing. As seen, when AEX counting is enabled, the logger adds an overhead of ≈1,076 ns per counted AEX. Tracing AEXs instead of just counting increases the overhead again by 1.04× per AEX.


[66] sgx_ecall_SSL_set_quiet_shutdown
[55] sgx_ecall_SSL_get_error
[23] sgx_ecall_ERR_peek_error
[0] sgx_ecall_SSL_read
[26] sgx_ecall_ERR_clear_error
[69] sgx_ecall_SSL_shutdown
[64] sgx_ecall_SSL_write
[54] sgx_ecall_SSL_do_handshake
[68] sgx_ecall_SSL_free
[27] enclave_ocall_write
[51] sgx_ecall_SSL_new
[37] enclave_ocall_execute_ssl_ctx_info_callback
[52] sgx_ecall_SSL_set_fd
[56] sgx_ecall_SSL_get_rbio
[26] enclave_ocall_read
[53] sgx_ecall_SSL_set_accept_state
[58] sgx_ecall_BIO_int_ctrl
[38] enclave_ocall_alpn_select_cb
1000
1000
1138
2000
1000
5138
2000
1000
1000
1000
1138
1000
138
1000
1000
1000
1000
2000
3276
17276
999
2000
2000
12000
3000
1000
2000
2138
2276
1000
1000
1000
1000
1000
1000
1000
Figure 5: nginx + TaLoS main enclave calls. Square nodes are ecalls, round nodes are ocalls. Solid arrows indicate direct parents, dashed arrows indirect parents. Numbers on edges indicate call count, numbers in brackets indicate call id.

5.2 Optimisation of Enclaves

To evaluate the data analysis part of *sgx-perf*, we took a look at different enclaves to see if they have problems that can be detected by *sgx-perf*. We took a look at enclaves from the following projects: (i) TaLoS [2] with nginx [13]; (ii) SecureKeeper [5]; (iii) SQLite [37] and Glamdring [25] partitioned (iv) LibreSSL.

5.2.1 TaLoS with nginx. TaLoS [2] is an enclavised LibreSSL [30] designed to be a drop-in replacement. It can be used by applications that use OpenSSL or LibreSSL to enhance their security by relocating all cryptographic operations into an enclave. TaLoS exposes the OpenSSL interface as its enclave ecall interface. We therefore tried to find out if the OpenSSL interface is suitable as an enclave interface or if performance issues can arise. As TaLoS is meant to replace OpenSSL in other applications, we used nginx [13] as a host application that calls into TaLoS. Our evaluation consists of performing 1000 HTTP GET requests with curl[38] against our TaLoS nginx server.

The enclave interface consists of 207 ecalls and 61 ocalls of which 61 and 10 were called 27,631 and 28,969 times, respectively. Overall, 60.78% of ecalls and 73.69% of ocalls were shorter than 10μs. We took a look at the main part of functions – that is accepting connections, reading, writing and shutdown. In nginx, this comprises the function calls seen in Figure 5. We can directly see many relationships

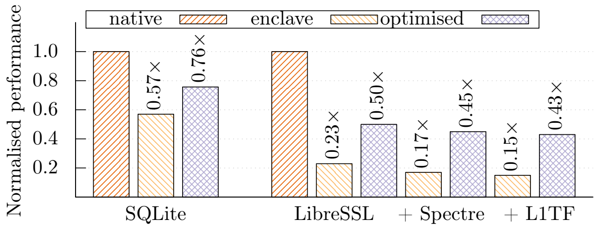


Figure 6: Normalised performance for SQLite and LibreSSL for native, enclavised and optimised execution.

that occur 1000 times (or a factor thereof) which corresponds to our 1000 requests. However, we can also see the first shortcoming of the OpenSSL interface, namely its error handling. OpenSSL does not directly return meaningful error codes in its functions but rather pushes errors into an error queue. Access to that queue is available through the `ERR_*` family of functions. This incurs additional enclave transitions compared to errors being directly returned by the function (ecalls 23, 26 in Figure 5).

Reading from and writing to the underlying socket is also not optimal. In TaLoS, the `read` and `write` system calls are implemented as ocalls which incurs a transition (ocalls 26 and 27 in Figure 5). While this is required as OpenSSL needs to communicate via the network to implement the TLS protocol, this has a non-negligible impact on the performance. A better design would be to batch the ocalls to `read` and `write` or give the application control of the socket and use OpenSSL's BIO abstraction layer to access it from inside the enclave. Unfortunately this requires changes to the implementation of the TLS protocol and to the calling application.

In summary, the OpenSSL interface is not suitable as an enclave interface due to its high number of transitions for simple operations. We analysed the code and found that while the authors of TaLoS already did a number of optimisations, the main blocker for more performance is the goal of being a drop-in replacement.

5.2.2 SQLite. Several research works have considered running an SQL database inside an enclave [3, 33]. We wrote a microbenchmark that performs a series of insert operations into a database persistently stored on disk, implementing system calls naïvely as ocalls. We ran experiments similar to those of the LibSEAL paper, replaying commits from popular git repositories, and achieved a performance of ≈23087 requests/s. The enclavised version achieved ≈13160 requests/s (0.57×). *sgx-perf* reported 41 ocalls, three of which are each responsible for 33% of the execution time: `lseek`, `write` and `fsync`.

On Linux, SQLite v3.23.1 makes separate calls to `lseek` and `write` in order to persistently store the database on disk. The `lseek` ocalls were quite short with an average duration of 4µs whereas the `write` ocalls took 17µs on average. The *sgx-perf* analyser showed a potential optimisation opportunity for the SDSC problem in the form of call merging. Merging the `lseek` and `write` calls lead to an increase to ≈17483 requests/s, 33% more, by eliminating one ocall. Figure 6 compares the results.

5.2.3 Glamdring. Glamdring [25] is a partitioning framework which aims to automatically partition applications into an untrusted and trusted part with the trusted part living inside an SGX enclave. The workflow of Glamdring looks as follows: First, the developer marks certain data as sensitive. Second, Glamdring employs static dataflow analysis and static backwards slicing to find all functions accessing and modifying the sensitive data. Lastly the application is partitioned and code is generated. Glamdring achieves 0.23× - 0.8× the performance of the native application.

We analysed a Glamdring-partitioned LibreSSL v2.4.2 and repeated the signing benchmark of the paper (signing certificates) with our logger attached. The benchmark runs for 30 seconds and tries to sign as many certificates as possible. The results show a performance of 33.88 signs/sec. Working set analysis showed a small enclave with 61 pages used after start-up and 32 pages used during benchmark execution.

The enclave interface consists of 171 ecalls and 3357 ocalls. In total, *sgx-perf* logged 18 ecalls being called 6.6 million times and 35 ocalls being called 110,511 times. Analysis showed that the `bn_sub_part_words` ecall is the main performance hog by accounting for 99.5% of all ecalls and a mean execution time of just 3µs which is basically the transition time. Therefore, this call's actual computation is too short compared to the transition time needed. This also applies to some other ecalls but these are called <1% of the time. The ocalls also show short execution times with 78.65% of all ocalls being shorter than 1µs (95.34% shorter than 10µs).

The *sgx-perf* analyser found multiple SNC and SISC problems, mainly short ocalls of the `BN_` family of calls for big number processing. Also, the `bn_sub_part_words` ecall was identified as an SISC problem. This ecall was marked for potential batching. Looking at the code, we could see that this call was always called in pairs inside `bn_mul_recursive`:

```c
void bn_mul_recursive(...) {
  // ...
  switch (c1 * 3 + c2) {
  case -4:
    ecall_bn_sub_part_words(t, a+n, a, tna, tna-n);
    ecall_bn_sub_part_words(t+n, b, b+n, tnb, n-tnb);
    break;
  // ... Repeated three more times
  }
  // ...
}
```

As the name suggests, the function is calling itself recursively at the end. By moving this entire function inside the enclave we were able to remove the successive ecalls to `bn_sub_part_words` and improve the performance by 2.16×.

We compared native LibreSSL against the original Glamdring version and our optimised version with less ecalls and ocalls. We also compared against applying the Spectre and Foreshadow (L1TF) microcode updates to see its impact on a real application. The normalised results can be seen in Figure 6. On our machine we see a higher native speed compared to the results from the paper (145 vs. 63 signs/s) but similar enclave performance (33 vs 36 signs/s). We attribute that to the difference in hardware, operating systems and compiler versions. As seen, optimising the automated partitioned code lead to a 2.16× speed-up and even an 2.66× (Spectre [6, 29]) and 2.87× (L1TF [42]) speed up on the patched system. This further underlines the need to reduce excessive enclave transitions and to have a good enclave interface.

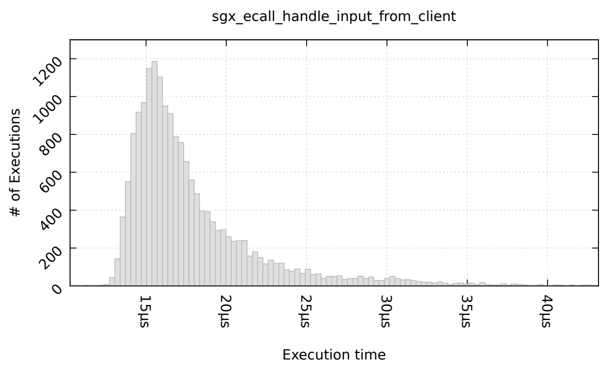


Figure 7: Generated histogram of call execution times for one of SecureKeeper's ecalls grouped into 100 bins.

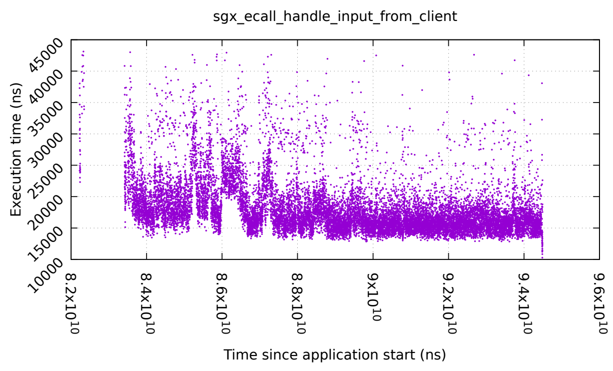


Figure 8: Generated scatter plot of call execution times for one of SecureKeeper's ecalls.

5.2.4 SecureKeeper. SecureKeeper [5] is a secure version of the Apache ZooKeeper [12] coordination service. It uses SGX to implement a proxy that sits between clients and ZooKeeper to store data transparently encrypted. Client-proxy communication is transport encrypted whereas the proxy en-/decrypts the payload an path of the packet going to/coming from ZooKeeper. This allows running the service in untrusted environments like cloud platforms. SecureKeeper's architecture only incurs an overhead of 11% compared to an unsecured ZooKeeper.

We analysed a single SecureKeeper instance running under full load for 31 seconds with our logger attached, similarly to the benchmarks shown in the paper. The logger recorded 1.1 million ecall and 111 ocall events. The enclave interface consists of just two ecalls and six ocalls of which two and three were called, respectively. Analysis showed that both ecalls have a mean execution time of ≈14µs and ≈18µs, ≈4-6× the transition cost.

SecureKeeper uses the SGX SDK's synchronisation primitives to coordinate access to queues and to a map. The map is only written when a client connects whereas a queue exists per client and is synchronised per client. During our testing we saw 18 synchronisation related ocalls which were issued during the connection phase of the benchmark in which all clients simultaneously connect, therefore creating high contention on the map. We observed low contention on the queue, as no ocalls were issued during actual benchmark execution. The remaining ocalls were debugging print ocalls during connection establishment.

In Figure 7 we can see the generated histogram for the ecall `sgx_ecall_handle_input_from_client`. It can be seen, that almost all calls are longer than 10µs with most calls taking about 15µs. In Figure 8 we can also see the call execution times plotted over the time of the application.

We were not able to spot any performance optimisation possibilities. The enclave interface is very narrow and no calls are short lived. Furthermore, SecureKeeper already uses some optimisations, e.g., saving ocalls for memory allocation by estimating the needed amount and allocating the memory before the ecall. As SecureKeeper is meant to run in a cloud environment, we looked at the enclave working set to determine how affected SecureKeeper might be by paging. Working set analysis showed 322 pages (1.26 MiB) are needed at start-up but during execution only 94 (0.36 MiB) are used. SecureKeeper spawns one enclave per client which explains the low usage as every client needs that many pages. The enclaves are sufficiently small, if SecureKeeper were able to fill up the whole EPC on its own, it could operate 249 enclaves in parallel without experiencing paging. We consider SecureKeeper sufficiently optimised with regards to the enclave interface and enclave size.

6 RELATED WORK

To our knowledge no comparable high-level analysis tool for SGX exists so far. As stated in §2.4, VTune Amplifier, a commercial low-level analysis tool from Intel, can inspect and profile SGX enclaves to find performance bottlenecks on an instruction level. The Linux tool `perf` [4] provides similar insights but does not offer support for SGX enclaves. That is to say, these tools report to the developer which instruction, line of code or function is costly and should be optimised to improve the performance of the system. However, contrarily to *sgx-perf*, they do not address the performance issues specific to Intel SGX: costly enclave transitions and paging.

SGX performance has been a topic of interest for various work. SCONE [1] and SecureKeeper [5] both measured the impact of SGX paging on an application exceeding the EPC size. They concluded that enclaves should never exceed the EPC size, as paging is simply too costly. Weisse et al. [44] and Zhao et al. [46] both looked at enclave transition performance and showed that those transitions are very costly. The number of transitions should therefore be reduced as much as possible. While they proposed solutions to minimise the impact of these problems, such as executing enclave transitions asynchronously [1] or using a custom memory allocator [31], they do not provide a tool to measure the impact of SGX-specific problems in an arbitrary application.

Gjerdrum et al. [8] were the first to present a list of SGX performance principles and recommendations for enclave developers in a cloud scenario. The authors do not directly recommend minimising enclave transitions but instead state that during an ecall the supplied data should be as small as possible to reduce the time it takes to copy it inside the enclave. While we agree, we think that minimising the actual number of transitions is more important. However, we disagree with their second recommendation stating that enclaves should not exceed 64kB in size to increase start-up times and prevent paging. While EPC memory is scarce, we argue that having an efficient strategy to minimise enclave paging is more important than limiting the size of the enclave, especially in a cloud environment where the EPC might already be oversubscribed.

While we are the first to propose a profiling tool specific for Intel SGX, the idea of profiling tools specific to a particular system is not novel. For example, LIKWID [40, 41] or MemProf [24] both use the low-level performance counters of modern processors (e.g., number of cache misses) to extract high-level metrics (e.g., memory bandwidth or remote accesses of memory objects on a NUMA machine) that help the developer to improve the performance of their application with new, more useful insights.

Performance anti-pattern detection is a research area that focuses on documenting common performance problems as well as their solutions. Smith and Williams [36] were the first ones to explore anti-patterns that have consequences on the performance of the system. They presented four anti-patterns: (i) excessive dynamic memory allocation; (ii) successive (database) operations; (iii) critical section of code where most of the processes cannot execute concurrently and have to wait; and (iv) wide variability in response time. Subsequently, Parsons et al. [32] and Cheng et al. [7] proposed new tools to automatically detect these performance anti-patterns in enterprise systems. The reader could view the problem we address as performance anti-pattern detection specific to Intel SGX.

7 CONCLUSION

Trusted computing with Intel SGX has become an important topic in the software development world. Several works [1, 5, 44, 46] have shown that paging and enclave transitions have a strong impact on the performance of the system. However, there is, to the best of our knowledge, no tooling support that gives an high-level overview of enclave behaviour to uncover potential performance problems.

In this paper we presented *sgx-perf*, a collection of tools that can trace enclave execution during runtime to generate a trace file. This file can then be analysed regarding different criteria to identify SGX-specific performance anti-patterns and to give developers hints to increase enclave performance.

We evaluated *sgx-perf* by analysing four SGX applications. Applying the recommendations given by *sgx-perf*, we were able to increase performance by 1.33× - 2.16×. The source code is available on GitHub[3].

ACKNOWLEDGMENTS

We thank our anonymous reviewers and our shepherd, Laurent Réveillère, for their helpful comments. This work was supported by the German Research Foundation (DFG) under priority program SPP2037 grant no. KA 3171/6-1 and the European Union's Horizon 2020 programme under grant agreement 690111 (SecureCloud).

REFERENCES

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[2] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2017. *TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves.* Technical Report. Imperial College London.

[3] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2018. LibSEAL: Revealing Service Integrity Violations Using Trusted Execution. In *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys).*

[4] Multiple Authors. 2018. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on 2018-05-18.

[5] Stefan Brenner, Colin Wulf, Matthias Lorenz, Nico Weichbrodt, David Goltzsche, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 15th International Middleware Conference (MIDDLEWARE).*

[6] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv:1802.09085* (2018).

[7] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-patterns for Applications Developed using Object-Relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE).*

[8] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. 2017. Performance of Trusted Computing in Cloud Infrastructures with Intel SGX. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER).*

[9] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Costa, Christof Fetzer, Pascal Felber, et al. 2018. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN).*

[10] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive* (2016).

[11] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. 2017. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet).*

[12] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems.. In *USENIX Annual Technical Conference (USENIX ATC).*

[13] Nginx Inc. 2018. Nginx. http://nginx.org/. Accessed on 2018-05-18.

[14] Intel. 2014. Intel Software Guard Extensions Programming Reference, Revision 2. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[15] Intel. 2018. Intel Software Guard Extensions Developer Reference for Linux OS. https://download.01.org/intel-sgx/linux-2.1/docs/Intel_SGX_Developer_Reference_Linux_2.1_Open_Source.pdf. Accessed on 2018-05-18.

[16] Intel. 2018. Intel Software Guard Extensions SDK for Linux. https://01.org/intel-softwareguard-extensions. Accessed on 2018-05-18.

[17] Intel. 2018. Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass (CVE-2017-5753) Side Channel Exploits. https://software.intel.com/sites/default/files/180204_SGX_SDK_Developer_Guidance_v1.0.pdf.

[18] Intel. 2018. Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed on 2018-05-18.

[19] Xiaojin Jiao. 2018. potential security issue: ecall SSL write using user check. https://github.com/lsds/TaLoS/issues/13.

[20] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203* (2018).

[21] R. Krishnakumar. 2005. Kernel korner: kprobes-a kernel debugger. *Linux Journal* (2005).

[22] Arseny Kurnikov, Klaudia Krawiecka, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2018. Using SafeKeeper to Protect Web Passwords. In *Companion Proceedings of the The Web Conference 2018 (WWW).*

[23] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys).*

[24] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: a Memory Profiler for NUMA Multicore Systems. In *USENIX Annual Technical Conference (USENIX ATC).*

[25] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference (USENIX ATC).*

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv:1801.01207* (2018).

[3] https://github.com/ibr-ds/sgx-perf

[27] LSDS Team, Imperial College London. 2018. github: sgx-lkl. https://github.com/lsds/sgx-lkl.

[28] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP).*

[29] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. 2018. github: spectre-attack-sgx. https://github.com/lsds/spectre-attack-sgx.

[30] OpenBSD Project. 2018. LibreSSL. https://www.libressl.org/. Accessed on 2018-05-18.

[31] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys).*

[32] Trevor Parsons and John Murphy. 2008. Detecting Performance Antipatterns in Com-ponent Based Enterprise Systems. *Journal of Object Technology* (2008).

[33] Vasily A Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rüdiger Kapitza. 2018. STANlite–a database engine for secure data processing at rack-scale level. In *Proceedings of the Sixth International Conference on Cloud Engineering (IC2E).*

[34] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. 2016. S-NFV: Securing NFV States by Using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security 2016).*

[35] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS).*

[36] Connie U Smith and Lloyd G Williams. 2001. Software Performance AntiPatterns; Common Performance Problems and Their Solutions. In *Int. CMG Conference.*

[37] SQLite Project. 2018. SQLite. https://www.sqlite.org/. Accessed on 2018-05-18.

[38] The curl project. 2018. curl. https://curl.haxx.se/. Accessed on 2018-05-18.

[39] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the Symposium on SDN Research (SOSR).*

[40] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 39th International Conference on Parallel Processing Workshops (ICPPW).*

[41] Jan Treibig, Georg Hager, and Gerhard Wellein. 2012. Best practices for HPM-assisted performance engineering on modern multicore processors. *arXiv:1206.3738* (2012).

[42] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium. (USENIX Security).*

[43] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security (ESORICS).*

[44] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA).*

[45] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (IEEE S&P).*

[46] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing. 2016. On the Performance of Intel SGX. In *3th Web Information Systems and Applications Conference (WISA).*

4 Security Issues Arising with the new Threat Model

When trying to increase the performance of an application, just optimizing the hot-path or the algorithm itself is not enough. With today's multicore hardware, an obvious technique for increasing performance is to utilize parallelism and multithreading to easily multiply performance numbers. Of course, multithreading not only improves performance when multiple cores can do work simultaneously but can also help in single-core situations when accesses to external, possibly slower, resources are in the hot path of, e.g., a request. In such a case, a thread can go to sleep, and a new thread can start working on the next request until the first thread can resume its work when the accessed resource is ready. However, multithreading enables a new class of attacks that need to be considered in the threat model: exploitation of synchronization bugs.

Studies have shown [57, 99], that writing correct multithreaded code is hard, and synchronization bugs are often overlooked when it comes to security. A synchronization bug causing unintended behaviour is dependent on specific thread scheduling which is controlled by the operating system. The chance of threads being scheduled so that the bug is triggered might simply be very low, therefore the bug might not trigger unless the thread scheduling is specifically altered to force the bug being triggered. The traditional threat model for remote applications considers attackers which are capable of sending arbitrary requests to a service. An altered thread scheduling might be achievable by a remote attacker by crafting and sending requests in a certain way; however, due to the remote nature of the attacker, doing so deterministically is very hard if not impossible as network latencies are typically not stable enough.

Local access is seldom considered as an attacker cannot be given administrative privileges as they could just read/write application memory, deny access, or similar. A restricted local access can be considered but increases the TCB to include the operating system and its isolation mechanisms. Such an attacker has a far better success chance as they could exhaust local resources to force the operating system scheduler into a specific behaviour, such as creating high processor contention or just keeping exclusive access to a required shared resource, e.g. hardware. This, however, does not allow deterministic exploitation of any synchronization bug, e.g., data races, which are very sensitive to timings.

With trusted computing, the threat model changes and the TCB is drastically lowered to only include the application and the necessary TEE components, e.g., in the case of

SGX the processor and its SGX implementation. Now, the attacker is assumed to have full control over the operating system as the TEE protects the application even in this scenario. This also includes full control over thread scheduling and therefore an attacker is capable of manipulating the thread scheduling in any way they deem beneficial. This makes it theoretically possible to trigger any synchronization bug by enforcing a specific thread schedule.

4.1 AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves

In *AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves* [96] (on page 43) it is shown that multithreaded enclaves that contain synchronization bugs can be exploited by manipulating the thread scheduling. AsyncShock offers a flexible way of forcing an application through a specific thread schedule by letting attackers write a *playbook* that details how threads should be scheduled. To influence the actual thread scheduling AsyncShock utilizes a technique first described by Xu et al. [100]. In their paper, it is shown how to extract secrets from an enclave with known code by tracing the order in which enclave pages are accessed. To do this, all enclave pages are stripped of their page permissions which causes execution to fault with a segmentation fault. The fault is then caught, the page access recorded and page permissions are restored for that page only so the execution can commence until the next fault. This fault is also caught, strips the permissions of the previous page again and grants permissions to the current page. With this technique, the whole execution can be traced.

Stripping page permissions from enclaves works because they are stored twice: once inside the normal, insecure page table and once inside the secure EPCM. As shown in Figure 4.1, accesses to enclave pages are also checked against both permissions with the page table being checked first. As the page table is stored in normal memory, it is possible for an untrusted application to edit the permissions stored in it.

AsyncShock utilizes the same described technique but instead of tracing execution, it uses the fault handler to stop execution for a specific thread until other threads have done a certain amount work before stopping those and resuming the first thread. When to stop and resume threads is recorded in the playbook which is read by AsyncShock on launch and then followed while executing the application under attack.

To demonstrate the effectiveness of AsyncShock, two different kinds of synchronization bugs were successfully exploited. The paper shows how a use-after-free bug in an example application that only exists due to missing synchronization can be exploited to allow redirecting the control flow of the enclave to another function inside the enclave. This attack was tested on two machines multiple times and AsyncShock had a 100% success rate in exploiting the enclave. The second bug, a Time Of Check To Time Of Use (TOCTTOU) bug, was harder to exploit as it relied on exact timing. AsyncShock, however, was able to

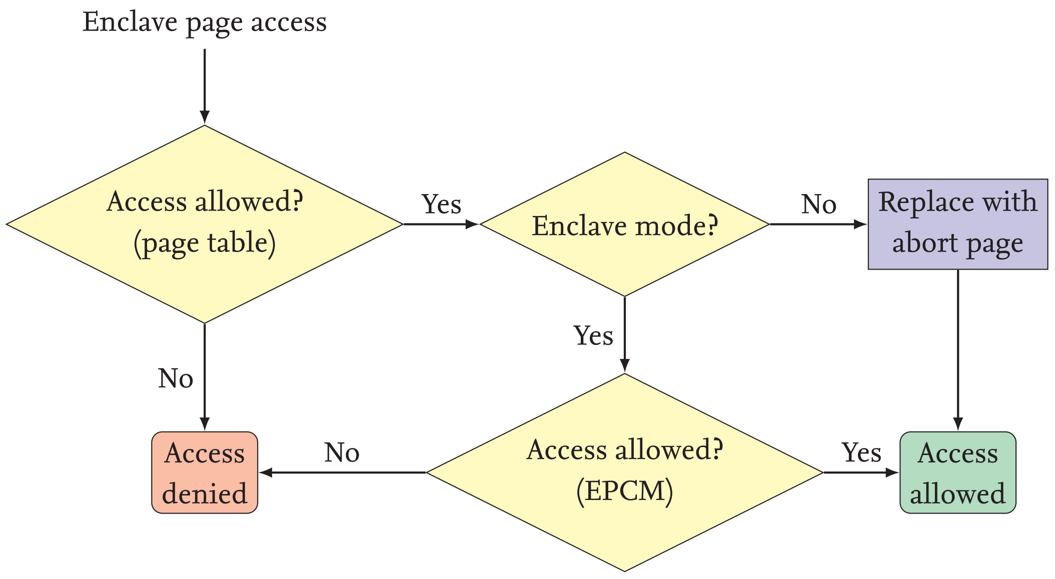


Figure 4.1: Memory access permission checks on an enclave page. Permissions are checked and managed twice: once by the MMU (page table) and once by SGX (EPCM).

achieve a near 100% success rate in this case.

In conclusion, AsyncShock shows that under the new threat model of TEEs synchronization bugs need to be taken serious as their exploitation can lead to an attacker gaining control of the enclave.

4.2 Related Work

While synchronization bugs with security implications have existed before SGX, their usefulness as an attack vector was typically low as the threat model does not include an attacker that has control over the thread scheduler. Such an attacker would basically have control over the operating system and therefore would have much more potent attack vectors at their disposal. Nevertheless, there exist studies on synchronization bugs and their security implications. Dean et al. [31] have shown that a TOCTTOU attack vector exists in the Linux kernel when performing file system access control decisions, how to exploit it and propose countermeasures. Borisov et al. [18] defeated the proposed countermeasures and Tsafrir et al. [88] improved the proposed countermeasures to make exploitation harder. As can be seen, fixing synchronization bugs is hard, especially when the Application Binary Interface (ABI), here in form of the system call interface, must not be modified to preserve backwards compatibility.

A more general study by Yang et al. [101] identified concurrency attacks and their risk on real-world systems. They identified that the exploitability of a concurrency error is dependent on the size of the timing window in which the error occurs, and that attackers might be able to enlarge that window through specially crafted inputs. Especially memory races have small attack windows at the level of nanoseconds. AsyncShock is able to widen the attack window significantly to enable reliable exploitation. Yang et al. propose solutions for better detection of such bugs in static analysis tools. They also analysed exploit defence techniques proposed by other authors that only assumed sequential execution and concluded that some simply fail to work under multithreaded execution or require potentially performance degrading changes.

AsyncShock's approach is similar to that of Xu et al. presented in *Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems* [100]. Here, page faults are used to trace enclave execution by removing enclave page permissions, recording page faults, briefly allowing execution and removing permissions again. This produces a trace of page fault addresses that can reveal what the enclave is working on when the enclavized software is known which in turn can reveal secrets. Xu et al. demonstrated their attack by enclavizing open source software for text and image processing and showed that they were able to reconstruct both secret text and images simply by observing page accesses. This is possible because the enclavized software uses different page access patterns depending on the input. AsyncShock uses the same technique of removing page permissions and utilizes it to enforce a certain thread schedule instead of tracing page accesses. This modified approach only requires modifying page permissions of a few pages near the critical section instead of the whole enclave.

Both AsyncShock and Controlled-Channel Attacks are limited by the page fault mechanism's granularity of 4 KiB pages of memory. This makes it hard to accurately stop execution on an enclave within a page. *SGX-Step* [90] circumvents this limitation by proposing a new way to interrupt an enclave: Timer interrupts. When an enclave is executing, interrupts still need to be handled. To achieve this, SGX can perform an AEX that leaves the enclave to handle the interrupt, see Section 2.1. SGX-Step works by configuring a timer inside the Advanced Programmable Interrupt Controller the delivers a high number of interrupts to the system. This continuously interrupts the enclave allowing SGX-Step to effectively single-step on an instruction level through the enclave and allows for fine-grained control over enclave execution.

Interrupting an enclave through page faults or timer interrupts is a disruptive process that slows down enclave execution. If only data exfiltration is needed and no control flow management, then a different page table based approach is possible. In *Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution* [93] together with Jo Van Bulck we have shown that the access and dirty bits of the page table can be used to trace enclave execution similarly to Controlled-Channel Attacks. Again, data exfiltration was shown to be possible, this time from a cryptographic library, because the library did use different page access patterns when operating on key material.

There also exist potential countermeasures against AsyncShock. The most obvious one is to make sure that the enclave does not contain synchronization bugs. This might be achievable by utilizing tools such as ThreadSanitizer [81] to detect data races. Völp et al. [94] propose a different solution to guard against a malicious OS that interrupts enclave execution at arbitrary times. They propose to utilize *delayed preemption*, inspired from the L4-family of microkernels, inside an enclave. This technique allows an enclave to disable preemption during execution of a critical section by deferring the preemption until after the critical section has completed execution. For AsyncShock, this would imply that it is no longer possible to preempt a thread at will. However, the proposed solution would require Intel to built it into the hardware which makes it a nice idea, but ultimately not possible to actually use.

There has also been work in the direction of static analyses and verification to prove that an enclave is not capable of leaking secrets. Moat [85] proposes to verify enclave code to be unable to disclose secrets by using static analyses and "ghost variables" to track the secrecy of sensitive data, similar to taint tracking. As the paper shows, the approach is promising, but their implementation does not take multithreaded enclaves into account.

A more realistic defence is proposed in Haven [16]. Haven proposes a library OS to run unmodified Windows applications inside SGX enclaves coupled together with a shield module. The shield module manages synchronization primitives to ensure correct behaviour which can help but still requires the application to actually use them correctly. Haven also proposes to decouple hardware threads from application threads and to switch to user-level scheduling inside the enclave. While this makes it harder to enforce a certain thread schedule, it does not make stopping user level threads impossible as AsyncShock can still interrupt them. If it were possible for a hardware thread to recover a stopped user-level thread by transferring its context to another hardware thread, AsyncShock would be unable to stop user-level threads. Haven, however, does not propose a user-level thread recovery mechanism, possibly because building such a mechanism might be too complex or even impossible with the state of SGX at the time.

In 2023, Intel released a new SGX extension called *Asynchronous Enclave Exit Notify* [41] based on the work on Constable et al. [29]. This extension allows an enclave to modify the behaviour of an AEX by changing how `ERESUME` is executed. An enclave thread that requests AEX-Notify set via a flag, will resume execution not at the point of interruption but instead again at the entry point defined in the TCS. The entry point can check whether it was invoked via `EENTER` or `ERESUME` and react accordingly. Constable et al. [29] use this new mechanism to defend against the aforementioned SGX-Step [90]. The possibility of changing `ERESUME` to restart at the predefined entry point, however, could make it possible to implement a user-level thread recovery mechanism. Despite that, the feasibility of this endeavour is out of scope for this thesis but should be explored in the future.

(Publication starting next page)

AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves

Nico Weichbrodt, Anil Kurmus†, Peter Pietzuch*, and Rüdiger Kapitza

TU Braunschweig, †IBM Research Zurich, *Imperial College London
{weichbr,kapitza}@ibr.cs.tu-bs.de,kur@zurich.ibm.com,prp@imperial.ac.uk

Abstract. Intel's Software Guard Extensions (SGX) provide a new hardware-based trusted execution environment on Intel CPUs using *secure enclaves* that are resilient to accesses by privileged code and physical attackers. Originally designed for securing small services, SGX bears promise to protect complex, possibly cloud-hosted, legacy applications. In this paper, we show that previously considered harmless synchronisation bugs can turn into severe security vulnerabilities when using SGX. By exploiting use-after-free and time-of-check-to-time-of-use (TOCTTOU) bugs in enclave code, an attacker can hijack its control flow or bypass access control.
We present *AsyncShock*, a tool for exploiting synchronisation bugs of multithreaded code running under SGX. AsyncShock achieves this by only manipulating the scheduling of threads that are used to execute enclave code. It allows an attacker to interrupt threads by forcing segmentation faults on enclave pages. Our evaluation using two types of Intel Skylake CPUs shows that AsyncShock can reliably exploit use-after-free and TOCTTOU bugs.

Keywords: Intel Software Guard Extensions (SGX); Threading; Synchronisation; Vulnerability

1 Introduction

Recently, Intel's Software Guard Extensions (SGX), a new hardware-supported trusted execution environment for CPUs, has reached the mass market[1]. Similarly to previous trusted execution environments such as ARM TrustZone [1], SGX allows the execution of applications inside *secure enclaves*, without trusting other applications, the operating system (OS) or the boot process. Unlike previous solutions, SGX supports hardware multithreading, which is a fundamental requirement for modern performant applications.

Secure enclaves reduce the overall trusted computing base (TCB) to essentially the TCB of the enclave. SGX by itself, however, cannot prevent vulnerable enclave applications from being exploited. Although it was initially assumed that only small tailored applications would be executed inside enclaves [11], a

[1] https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf

recent trend is to consider enclaves as a generic isolation environment for arbitrary applications: VC3 [21] uses enclaves to secure computation for the Hadoop map/reduce framework; Haven [2] places a library OS inside an enclave for running unmodified Windows server applications.

This trend towards more complex, multi-threaded applications inside enclaves opens up new attacks. In particular, existing applications are designed to protect against a threat model that is not the same as the one for enclave code—traditional applications assume that the OS is trusted. As recent work has shown, an untrusted OS enables powerful side channel [28] and Iago [5] attacks.

In this paper, we explore a new angle for mounting attacks against SGX enclaves. We show that *synchronisation bugs* that are unlikely to be exploitable outside of SGX become reliably exploitable by carefully scheduling enclave threads. We achieve this by manipulating the page access permissions of enclave pages to force segmentation faults that interrupt enclave execution. Through this method, we are able to widen the traditionally small attack window of synchronisation bugs and increase the chances of a successful exploit.

Typically, the impact of such *concurrency attacks* [29] is to prevent or slow down certain activities in favour of others, create inconsistencies, extract data, bypass access control, or hijack the control flow of the attacked program (e.g., CVE-2009-1837, CVE-2010-5298, CVE-2013-6444). In the case of SGX, the impact of controlling code execution within an enclave is higher. At the time of writing, Intel only licenses the creation of SGX production enclaves after examination of the software development practices of the licensee[2]. Controlling enclave code execution would be a way to circumvent this practice, similarly to how "jailbroken" iPhones can execute non-Apple approved applications.

The contributions of the paper are:

- we show that synchronisation bugs are easier to exploit within SGX enclaves than in traditional applications. This is partly because, by design, the attacker can control thread scheduling of enclaves in the SGX attacker model;
- we describe AsyncShock, a tool that facilitates the reliable and semi-automated exploitation of synchronisation bugs in SGX enclaves. AsyncShock leverages the ability of the untrusted OS to arbitrarily interrupt and reschedule enclave threads. AsyncShock is designed to target enclaves built with the official SGX Software Development Kit (SDK) for Linux[3];
- we explain how to track enclave execution near critical sections by removing permissions from pages, which triggers notifications when enclave execution has reached a particular point;
- we show how use-after-free and TOCTTOU [3] bugs can be exploited by AsyncShock; and
- we provide evaluation results of attack success rates by AsyncShock on current Intel Skylake CPUs, exploring a variety of different implementations of the attack.

[2] `https://software.intel.com/en-us/articles/intel-sgx-product-licensing`
[3] `https://software.intel.com/en-us/blogs/2016/04/11/intel-software-guard-extensions-sdk-for-linux-availability-update`

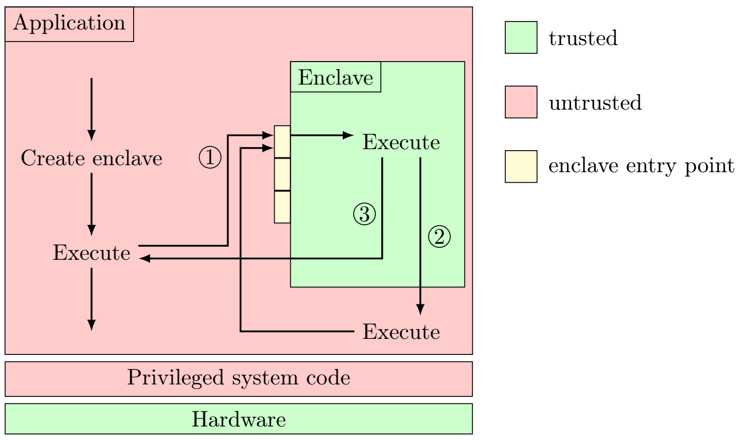


Fig. 1. Basic enclave interaction showing an `Ecall` ① into an enclave, an `Ocall` ② and the return ③ to the untrusted application. (The enclave entry points are shown in yellow.)

The paper is structured as follows: §2 provides background on SGX, the assumed attacker model and the impact of synchronisation bugs when using SGX; §3 describes our forced segmentation fault approach and the AsyncShock tool; §4 gives evaluation results and discusses protective measures; §5 surveys related work on SGX and similar attacks; and §6 concludes the paper.

2 Background

First, we give a brief introduction to trusted execution as implemented by SGX. After that, we present an attacker model that is tailored towards typical usage scenarios of SGX. Finally, we discuss the impact of synchronisation bugs.

2.1 SGX in a Nutshell

SGX allows developers to create an isolated context inside their applications, called a secure *enclave* [18, 12]. Enclaves feature multiple properties: (i) enclaves are isolated from other untrusted applications (including higher-privileged ones) through memory access control mechanisms enforced by the CPU; (ii) memory encryption is used to defend against physical attacks and to secure swapped out enclave pages; and (iii) enclaves support remote attestation at the level of enclave instances.

Programming Model. A typical workflow for using SGX with the support of the SGX SDK [13] starts with creating an enclave as part of an application. The necessary instructions for creating an enclave are only callable from kernel mode (ring 0) and thus require kernel support. Once successfully performed, the application can issue `Ecalls` ① to enter an enclave as seen in Figure 1. Inside the

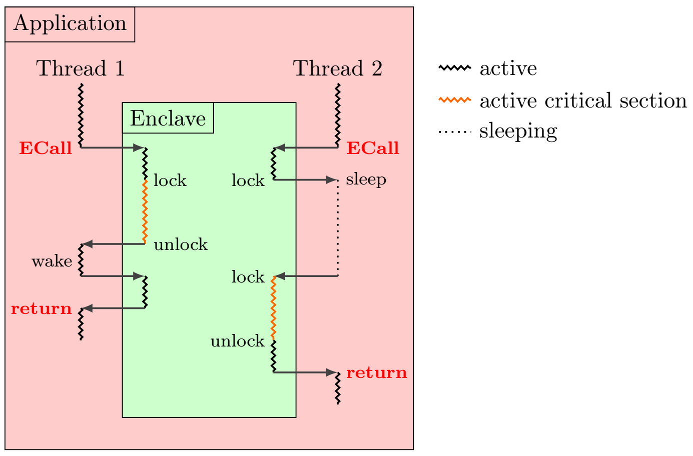


Fig. 2. Mutex lock/unlock operations provided by the SGX SDK may exit the enclave.

enclave, input parameters passed with the call can be processed, and enclave code is executed. Developers specify the enclave interface and the direction of data with a SDK-specific file written in the Enclave Description Language (EDL) [13]. The SDK handles data movement across the enclave boundary by performing the necessary memory copy operations. However, this is only supported for primitive data types and flat structures. Data structures with pointers are not deep-copied and therefore expose the enclave to TOCTTOU attacks.

`Ocalls` ② may be performed to leave the enclave and execute untrusted application code before an `Ecall` returns ③ to the enclave. While the enclave has access to inside and outside memory, the untrusted application is not allowed to access memory inside the enclave: any attempt to read enclave data results in abort page semantics, i.e. reading `0xFF`; write attempts are simply ignored.

Memory Management. Enclave creation and its memory layout are handled by an SGX kernel module. During enclave creation, the enclave code and data are copied page-by-page into the Enclave Page Cache (EPC), which is protected system memory. Mapped pages and their permissions are saved in the Enclave Page Cache Map (EPCM). Enclave page permissions are thus managed twice, once through the OS page table and once through the EPCM. Accessing an enclave page also leads to two permissions checks: once by the Memory Management Unit (MMU) reading the permissions from the page table, and once by SGX reading them from the EPCM. While it is possible to restrict page table permissions further using `mprotect`, it is not possible to extend them because the EPCM cannot be modified. The possibility of removing page permissions is important for AsyncShock—it means that an attacker can mark pages and get notified when they are used.

Support for Multithreading and Synchronisation Mechanisms. Each enclave must have at least one entry point that defines an address at which the enclave may be entered. The SDK implements a trampoline to allow multiple `Ecalls` through a single entry point. Multithreading is supported by having multiple entry points and permitting multiple threads to enter them concurrently. Similar to regular applications, interrupts may occur during enclave execution and must be handled. SGX achieves this by performing an Asynchronous Enclave Exit (AEX), which saves the current processor state into enclave memory, leaves the enclave and jumps to the Interrupt Service Routine (ISR). Enclave execution is resumed after the ISR finishes, restoring the saved processor state.

The SGX SDK offers synchronisation primitives such as mutexes and condition variables. These primitives do not operate exclusively inside the enclave: for instance, thread blocking requires a system call that is unavailable inside enclaves. Furthermore, managing a lock variable outside of the enclave is not advised because an attacker could change it. A hybrid approach has been adopted by Intel in which the lock variables are maintained inside the enclave whereas system calls are issued outside. Therefore, using synchronisation primitives may result in enclave exits. Figure 2 shows this behaviour for a mutex lock operation.

2.2 Attacker Model

We consider a typical attacker model for SGX enclaves: an attacker has full control over the environment that starts and stops SGX enclaves. They have full control of the OS and all code invoked prior to the transfer of control, using `Ecalls`, to the SGX enclave, and also when an enclave calls outside code via `Ocalls`. More specifically, the attacker can interrupt and resume SGX threads (see §2.1), which is the main attack vector exploited in this paper.

The attacker's goal is to compromise the confidentiality or integrity of the SGX enclave. For example, they may want to gain the ability to execute arbitrary code within the enclave. Note that we ignore availability threats, such as crashing an enclave: the untrusted OS can simply stop SGX threads.

2.3 Synchronisation Bugs in Software

Synchronisation bugs are caused by the improper synchronised access of shared data by multiple threads, and previous studies have shown that they are a widespread issue [15, 27]. A large number of tools were proposed to help developers find different kinds of synchronisation bugs, such as *atomicity violations* [6, 8, 16], *order violations* [9, 17, 32] and *data races* [20, 31]. These studies, however, do not explore the security implications of discovered bugs—in most cases, the discovered bugs lead to memory corruption or crashes. Although such bugs may seem benign and unlikely to occur, synchronisation bugs are likely to lead to exploitable security vulnerabilities [26, 7, 23].

Unlike traditional applications, in the context of SGX, enclave code is trusted both by its developper and Intel to run untampered on untrusted machines (e.g., hosted at an untrusted cloud service provider). Memory corruption inside an enclave may therefore be used to hijack execution of the enclave, potentially leading to the disclosure of enclave cryptographic keys. In addition, such vulnerabilities

```
...
free(pointer);
// thread is not active
// between those two
// statements
pointer = NULL;
...
```

Thread 1

```
...
...
if (pointer != NULL) {
   // use pointer
}
...
...
```

Thread 2

Fig. 3. Simple use-after-free. Thread one frees `pointer` but is not able to set it to `NULL` because thread two is scheduled in.

may be used by malicious attackers, e.g., botnet herders, to bypass Intel's vetting process and design rootkits that run inside the enclave and are undetectable by security software running in the OS: by design, the OS cannot introspect an enclave running in production mode. Therefore, vulnerabilities in enclaves are worrisome to enclave developpers, enclave hosters, and Intel.

In the following, we show that synchronisation bugs are a real security threat to enclave developers by exploiting two examples of the common atomicity-violation bugs: a use-after-free bug as well as a TOCTTOU bug.

3 Exploiting Synchronisation Bugs with Scheduler Control

Exploiting synchronisation bugs inside an SGX enclave can be broken down into: (i) finding an exploitable synchronisation bug; (ii) providing a way to interrupt and schedule enclave threads; and (iii) determining experimentally *when* to interrupt and schedule enclave threads. Next we describe each of these steps through the example of a use-after-free bug. In addition, we describe the AsyncShock tool, which generalises this approach and allow the easy adaptation of these steps to other vulnerabilities. We explain how AsyncShock exploits a TOCTTOU bug.

3.1 Exploiting Synchronisation Bugs inside an Enclave

We focus on the atomicity-violation class of bugs and show how such a bug can be exploited. Figure 3 shows an example of an atomicity violation. A possible use-after-free bug occurs if the first thread is interrupted directly after the `free` but before the assignment. The second thread performs a `NULL` check during this time, which succeeds even though the pointer has been freed. The call to `free` and the assignment were intended to be an atomic block by developer, but this is not reflected in the implementation.

During execution, such an interruption is a scheduling decision by the OS, and the probability that the interruption occurs at the right point is low. Furthermore, the thread itself is not paused but is scheduled again later while the second thread is still executing. The second thread may thus be interrupted during its execution before the freed pointer can be used.

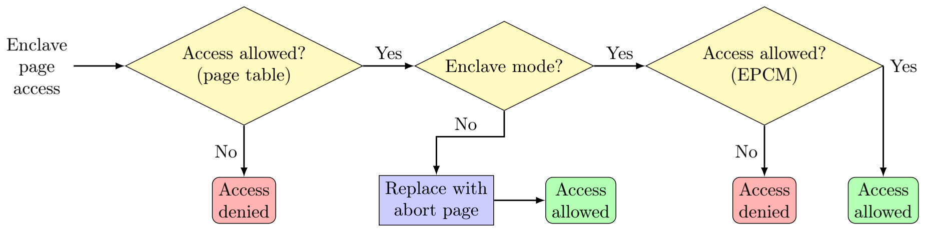


Fig. 4. Memory access permission checks on an enclave page. Permissions are checked and managed twice: once by the MMU (page table) and once by SGX (EPCM).

As shown in the litterature [29, 25], the attack window for memory races is small in practice. In some cases, the attacker may only have a single chance to exploit the vulnerability. Even if an attacker can execute the application many times, it may still take a long time until the interruption occurs at precisely the correct time. Being able to increase the attack window would thus help exploit such bugs more effectively. The AsyncShock tool aims to help exploit synchronisation bugs that are present inside an enclave by pausing and resuming threads during execution, which is possible when threads are inside an SGX enclave. We explore two techniques for interrupting threads, as described in the following sections.

3.2 Interrupting Threads via Linux Signals

One approach to interrupt threads is to leverage the Linux signal subsystem. Handling a signal interrupts the thread and redirects control to the signal handler. We therefore register a signal handler for the `SIGUSR1` and `SIGUSR2` signals. We use the `SIGUSR1` signal to pause a thread and the `SIGUSR2` signal to resume it again. A control thread sends these signals to specific threads based on a configurable delay. Elapsed time since the application start is measured and compared to the delay in a loop. When the delay is reached, a signal is issued. The signal is sent by the `pthread_kill` function provided by `pthreads`.

Pausing the thread is performed by using a condition variable to wait inside the signal handler that suspends the thread. Sending the resume signal causes a second signal handler invocation, which in turn uses a condition variable signal to resolve the wait in the first signal handler's invocation. Each thread has its own condition variable, facilitating the pausing and resuming of multiple threads.

While this approach works, it is unreliable and depends on the specifics of the Linux task scheduler. We experimented with different delays for the same exploit but observed the same success rate regardless of the delay. We suspect that the signal dispatching is too slow, leading to inaccurate interruptions. Furthermore, this approach requires a deterministic runtime of the program because the delay is fixed—non-deterministic execution inside the enclave defeats this approach.

3.3 Interrupting Threads via Forced Segmentation Faults

We explore another approach based on interrupting threads to force segmentation faults. Using `mprotect`, we remove the "read" and "execute" permissions from enclave pages, i.e. *marking* the page. As soon as an enclave page with stripped permissions is accessed, a `SIGSEGV` signal is dispatched by the kernel as a response to the fault generated by the MMU, notifying the attacker of the page access.

This approach exploits the fact that memory access checks with SGX are performed twice, as shown in Figure 4. The call to `mprotect` changes the permissions inside the page table, but not inside the EPCM. Therefore the access fails at the page table check, even though the real permissions are unchanged.

We install our own signal handler, as described in §3.2, but this time for `SIGSEGV`. Inside the handler, we can restore the page permissions, start a timer with a configurable delay and resume execution. If a timer is started, it can remove the permissions upon expiration. This leads to another `SIGSEGV`, which again invokes our handler. We can now employ the same thread stopping mechanism described for the signal approach using condition variables. The `mprotect` approach is more reliable than the signal approach because page permissions are changed instantaneously.

3.4 AsyncShock Tool

AsyncShock incorporates the described approaches into an easy-to-use tool. It is implemented as a shared library, which is preloaded using the `LD_PRELOAD` mechanism of the dynamic linker. To interact with the target application, AsyncShock provides its own implementation of certain functions that shadow their real implementations. An example is `pthread_create`, which is normally provided by the C standard library. AsyncShock provides its own implementation that observes thread creation and takes actions upon the creation of specific threads.

To use AsyncShock, an attacker must know how the scheduling needs to be influenced to successfully trigger an exploit. They then must transform the attack into a series of actions in reaction to certain events. Possible events include thread creation, segmentation faults and timer expirations; possible actions include pausing or resuming a thread, starting a timer or changing page permissions. We call this series of actions the attack *playbook*. AsyncShock enforces that the targeted application behaves according to the playbook while also manipulating the environment.

A textual representation of a playbook for the use-after-free bug from Figure 3 is given in Listing 1.1. It includes the definition of four reactions to events: on thread creation of the first thread, an enclave page (enclave base address + `0x5000`) is stripped of its read and execute permissions. By using `objdump`, we find that the `free` function is located on this code page, and we mark it to get notified when it is called. As soon as a thread calls the `free` function, a segmentation fault occurs, which is handled by the signal handler registered by

```
on thread creation "thread1":
   remove read,exec on enclave+0x5000

on thread creation "thread2":
   pause thread this

on segfault 1:
   set read,exec on enclave+0x5000
   remove read,exec on enclave+0x1000

on segfault 2:
   set read,exec on enclave+0x1000
   resume thread thread2
   pause thread this
```

Listing 1.1. Example playbook for the use-after-free bug from Figure 3.

AsyncShock. It reapplies the removed permissions and removes the permissions at another page. The marked page contains the calling function that we mark to get notified when `free` finishes.

The resulting segmentation fault is again handled by AsyncShock. This time, the faulting thread is paused, and the second thread is allowed to continue. As a result, the attack window has been widened for the second thread to exploit the bug.

3.5 AsyncShock in Action

We use AsyncShock to successfully exploit a use-after-free bug inside an enclave and take control of the instruction pointer. Listing 1.2 shows the exploited enclave code.

The code contains two `Ecalls`, one set-up `Ecall` only executed once and another `Ecall`. While the enclave contains no threads, the second `Ecall` is used by two untrusted threads to enter the enclave simultaneously. However, a synchronisation bug exists between lines 26 and 27 if multiple threads execute the `Ecall` function in line 19. `glob_str_ptr` is a shared variable between all executions that is freed inside the `Ecall` and set to `NULL`. The bug triggers if a thread has just executed the `free` but not yet the assignment, while a second thread enters the `Ecall` function. Due to the nature of the memory allocator provided by the SDK, the `malloc` call (line 20) provides the old `glob_str_ptr` address, which leads to `glob_str_ptr` and `my_func_ptr` pointing to the same memory. The second thread passes the `NULL` check and copies the user provided input to `glob_str_ptr`, which sets `my_func_ptr`. The function call in line 25 now receives its address from the user-provided input and can be given the address of another enclave function, thus hijacking the control flow inside the enclave.

We use AsyncShock with a playbook similar to the one shown in Listing 1.1 to exploit the bug. Figure 5 shows how AsyncShock exploits the bug in detail. AsyncShock lies dormant until one of its overwritten functions are called. The application first creates a thread that is paused immediately by AsyncShock ①.

```c
char *glob_str_ptr;

int other_functions(const char *c) { /* do other things */ }

int puts(const char * c) {
   printf("%s", c);
   return 0;
}

struct my_func_ptr {
   int (*my_puts) (const char *);
   char desc[8];
} my_func_ptr;

void ecall_setup() {
   glob_str_ptr = malloc(sizeof(struct my_func_ptr));
}

void ecall_print_and_save_arg_once(void *str) {
   struct my_func_ptr *mfp = malloc(sizeof(struct my_func_ptr));
   mfp->my_puts = puts;
   if (glob_str_ptr != NULL) {
      memcpy(glob_str_ptr, (char *)str, sizeof(glob_str_ptr));
      glob_str_ptr[sizeof(glob_str_ptr)] = ’\0’;
      mfp->my_puts(glob_str_ptr);
      free(glob_str_ptr);
      glob_str_ptr = NULL;
   }
   free(mfp);
}
```

Listing 1.2. Example enclave containing a user-after-free bug.

A second thread is created that is allowed to execute ②. At this point, the “read” and “execute” permissions are removed from the code page containing the `free` function. The second thread enters the enclave and begins execution. When it calls `free` ③, an access violation occurs, resulting in an AEX and a segmentation fault caught by AsyncShock ④. The permissions are restored for this page, but removed for another before the thread is allowed to continue.

When the next marked page is hit ⑤, resulting in another AEX and segmentation fault ⑥, we know that `free` has returned. In the signal handler, the permissions are restored again. We stop the thread and signal the sleeping thread to execute ⑦. This concludes the successful exploit.

3.6 AsyncShock and a TOCTTOU Bug

To show how AsyncShock can be adapted to a different type of bug, we exploit a TOCTTOU bug. Listing 1.3 shows an enclave with three `Ecalls`: two threads enter the enclave, the first through the `ecall_writer_thread` and the second through the `ecall_checker_thread Ecall`. The second thread checks (line 20) if the shared variable `data` contains the string `"bad data"` and, if so, does not access it. Other content leads to a successful check and results in the second use of the variable. The first thread writes to the shared variable after executing a non-deterministic amount of time.

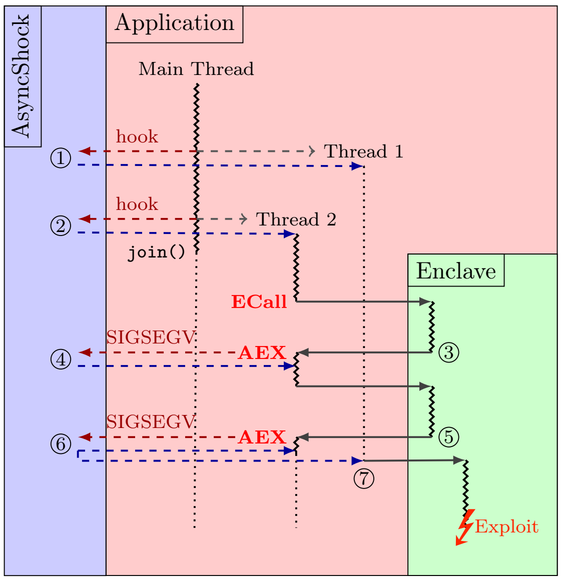


Fig. 5. AsyncShock exploiting the synchronisation bug from Listing 1.2

A TOCTTOU bug exists in lines 18 (check) and 19 (use). AsyncShock exploits this bug by delaying the writer thread, interrupting enclave execution after the check and then letting the writer thread proceed with the write to the shared variable. Interrupting between the check and the use in this example is challenging because the code pages containing `strncmp` and `memcpy` also contain some frequently called methods of the SDK. We therefore opt to start a timer right before entering the enclave, which expires between the check and the use. The timer has a configureable delay that postpones its execution. The correct delay must be determined empirically by observing the behaviour of the application with different delays. In our example, we observe the most successful exploits by choosing delays between 80000 and 120000 cycles, as described in §4.3.

4 Evaluation

To show the effectiveness of AsyncShock, we evaluate it by exploiting two atomicity violation bugs. First, we describe our evaluation set-up. After that, we present the results of exploiting a use-after-free bug and a TOCTTOU bug inside an enclave. We finish with a discussion of possible defenses.

4.1 Experimental Set-up

We evaluate the effectiveness of AsyncShock by exploiting a use-after-free bug, as well as a TOCTTOU bug, on real SGX hardware. We used a Dell Optiplex 7040 with an i7-6700 Intel CPU and 24 GB of memory. We also evaluate

```c
static char data[] = {'g', 'o', 'o', 'd', ' ', 'd', 'a', 't', 'a', '\0'};
static int random_wait = 0;

void ecall_setup() {
   random_wait = get_random_int();
}

void ecall_writer_thread() {
//This function has a constant delay >> check + use plus a random delay
//to simulate complex execution that takes a non-deterministic amount of time
   for (int i = 0; i < 100000; ++i);
   for (int j = 0; j < random_wait; ++j);
   snprintf(data, 10, "bad data");
}

int ecall_checker_thread() {
   char *str = calloc(1, 10);
   if (strncmp("bad data", data, 9) != 0) {
      memcpy(str, data, 10);
      printf("Access ok: %s\n", str);
      free(str);
      return 0;
   } else {
      printf("Sorry, no access!\n");
      return -1;
   }
}
```

Listing 1.3. Example enclave containing a TOCTTOU bug.

AsyncShock on a white-box server with an Intel E3-1230v5 CPU and 32 GB of memory. Both CPUs have four cores and are capable of hyper-threading, doubling the possible active threads. For our evaluation, hyper-threading has not been disabled. The desktop machine runs Ubuntu Linux 14.04.3 Desktop with kernel version 3.19.0-49; the server machine runs Ubuntu Linux 14.04.4 Server with kernel version 3.13.0-85. The server machine has a lower base load because fewer processes exist due to the missing desktop environment. All evaluations use a pre-release version of the SGX SDK which Intel provided to us.

4.2 Exploiting a Use-After-Free Bug

First, we establish a baseline by running the application without AsyncShock. We execute the application with its enclave one million times without observing a single successful exploit. We conclude that the attack window is too small to be exploitable just through controlled input.

We exploit the bug shown in Listing 1.2. Given the playbook from Listing 1.1, we can reliably exploit the use-after-free bug. We also modify the playbook to change the function arguments for the second thread so that the use-after-free results in a control flow modification, i.e. a call to `other_function`, which is otherwise not called. We execute the exploit 100000 times on both machines and observe a 100% success rate.

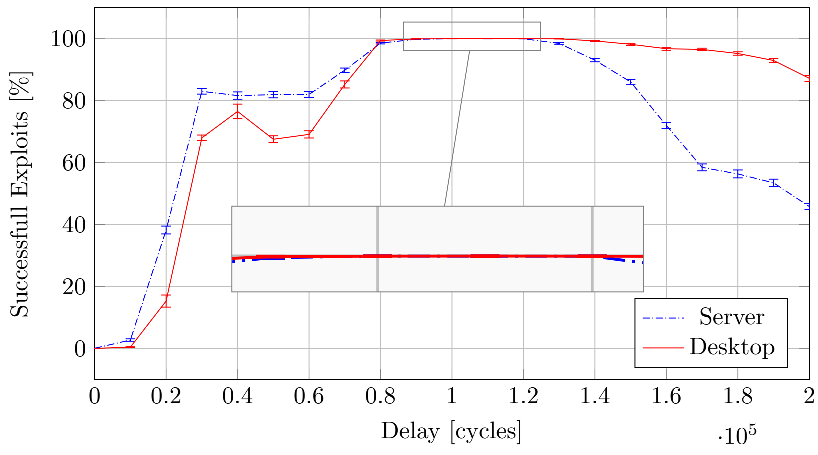


Fig. 6. Graph showing the success rate of AsyncShock exploiting the TOCTTOU bug at different timer delays.

4.3 Exploiting a TOCTOU Bug

To put the high rate chance of exploiting the use-after-free bug into perspective, we also consider a more difficult bug to exploit reliably: a TOCTTOU bug inside an enclave. Here we exploit the enclave code shown in Listing 1.3. We also establish a baseline by executing the application without AsyncShock. As with the use-after-free bug, we also do not see a single exploit occurring by chance. The non-deterministic delay in the writer thread is long enough so that the other thread can always perform the check and the use on the same data.

Next, we try to exploit the bug with AsyncShock. We evaluate a wide range of delays for the timer, as described in §3.6. Each delay is executed 10000 times. We record the successful exploits every 100 executions, obtaining 100 result sets per delay. We report the mean success rate for a given delay, with error bars representing a 95% confidence interval.

Figure 6 shows the results for the TOCTTOU exploit. As can be seen, the success rate varies not only with timer delay, but also differs for both machines with the same delay. We attribute this behaviour to the differences in base load and active processes on both machines. We are able to achieve near 100% success rates with timer delays of 80,000 cycles to 120,000 cycles. (As explained in §3.3, the delay is the time until AsyncShock removes the "execute" permissions from an enclave page, effectively forcing a stop to execution.) Our goal is to stop the enclave between the check and the use, which we achieve almost always with the correct delay.

Table 4.3 shows the results in more detail for selected delays. With a delay of 100,000, AsyncShock can almost always exploit the TOCTTOU bug with a low deviation. In conclusion, AsyncShock can be used to reliably exploit atomicity violation bugs with a high success rate.

Delay	Success rate	
(cycles)	Server	Desktop
80,000	98.57% ± 0.27	99.40% ± 0.17
90,000	99.81% ± 0.08	99.93% ± 0.05
100,000	99.99% ± 0.02	99.99% ± 0.03
110,000	99.98% ± 0.03	99.99% ± 0.02
120,000	99.98% ± 0.03	99.98% ± 0.03

Table 1. Deatiled success rates for selected delays.

4.4 Protective Measures Against AsyncShock

Our experimental results show that synchronisation bugs can lead to viable attacks against SGX enclaves. However, there already exist defense mechanisms for protecting from these attacks.

A first defence against the use-after-free bug is the sanitisation of user input as AsyncShock changes the `Ecall` parameters to direct the control flow. In general, sanitisation is advisable when unexpected input can be abused in a similar way to Iago attacks [5]. Enclave code should always check outside input for validity as an attacker may change the result from `Ocalls` or the parameters to `Ecalls` when using the SDK. In addition, enclave developers should not rely on the SDK's ability to defend against simple TOCTTOU attacks. While the SDK does copy `Ecall` parameters into enclave memory before passing them to enclave functions, it does not deep-copy data structures. Pointers in data structures are not followed and may lead to an enclave accessed outside memory. This type of vulnerability has often been exploited in OS kernels (e.g., [14] for Windows, and in general in filesystems [25]).

Another defense against the use-after-free bug presented here is possible because the bug relies on the in-enclave implementation of `malloc` to return recently freed memory. The attack can be mitigated by heap hardening methods, such as the one recently implemented in Internet Explorer through delayed free [10], or even with tools such as AddressSanitizer [22] that delay the reuse of recently freed memory or by changing the behaviour of the in-enclave memory allocator.

Protection from all synchronisation bugs can be achieved by prohibiting threading altogether—if only a single thread can enter the enclave at any time, no inconsistencies are possible due to serial execution. Such a solution, however, negatively impacts performance. If parallelism is needed, one can also adapt other techniques to work inside enclaves such as Stable Multithreading [30] or use tools such as ThreadSanitizer [23] during development in order to find and eliminate synchronisation bugs.

While many hardening techniques are applicable to enclave code, some traditional techniques do not work in the context of SGX. For example, the use of address space layout randomisation (ASLR) [19] is not directly applicable inside enclaves because any changes of the enclave memory would change the enclave measurement and therefore fail the signature check.

5 Related Work

Because SGX is a new technology with limited production use, only few use cases have been described so far. Haven [2] executes unmodified Windows applications inside an enclave. To achieve this, the combination of a shield module and a library OS provides the necessary execution support. The shield module manages synchronisation primitives and ensures their correct behaviour, similar to the SGX SDK. Furthermore, Haven tries to defend against Iago attacks be sanitising and checking the parameters of `Ecalls` and results of `Ocalls`. Haven also proposes a decoupling of enclave threads and host threads via user-level scheduling to hinder the exploitation of synchronisation bugs. However, AsyncShock should still be effective as it marks pages in close proximity to the synchronisation bugs to force an AEX. Thus, it does not necessarily need to observe the enclave-internal thread scheduling.

Fine-grained page tracking can be used for powerful side channel attacks [28]. For example, a JPEG image generated inside an enclave could be reconstructed outside: by paging out enclave pages to repeatedly induce page faults, memory accesses could be related to certain code paths. In contrast, AsyncShock is geared towards the exploitation of synchronisation bugs, albeit it can also be used to extract information from an enclave. However, for synchronisation bugs, AsyncShock only needs a small number of marker pages to track the enclave execution close to the critical section.

Yang et al. [29] identify concurrency attacks as a risk to real-world systems. They classify different types of attacks based on memory access patterns, and identify the attack window as an important factor for exploitability. Memory races usually have a small attack window at the level of nanoseconds. AsyncShock widens the attack window by stopping threads when a critical state is reached, steering other activities to allow reliable exploitation of memory-based concurrency bugs.

Synchronisation bugs have also been studied for their security implications. For instance, TOCTTOU races often affect filesystem-related code, typically when performing access control decisions. Dean and Hu [7] propose a countermeasure to alleviate those risks. Borisov et al. [4] show that this probabilistic countermeasure can be reliably defeated with filesystem mazes. Tsafrir et al. [25] propose another way to instrument those access checks to make the exploitation of those races significantly more difficult even against filesystem mazes.

Twiz and Sgrakkyu [26] extensively treat techniques for the exploitation of logical bugs in OS kernels. Jurczyk and Coldwind [14] describe how to exploit race conditions via memory access patterns in the Windows kernel. The Windows kernel copies the arguments to system calls from user to kernel space. However, the kernel does not copy pointer-referenced data in some cases. The authors exploit this by using the Bochs CPU emulator to interrupt the kernel, similar to how AsyncShock swaps out the data between two reads by the kernel—a classical TOCTTOU attack. However, in contrast, AsyncShock attacks an SGX enclave and not the kernel, in a setting where the attacker controls the scheduler and has reliable side channels on a thread's progress.

Moat [24] makes a first step towards the verification of SGX enclaves. The authors propose an approach to verify that enclave code is unable to disclose secrets. They employ static analysis on the x86 machine code, introducing "ghost variables" to track the secrecy of data in a manner similar to taint tracking. With this method, they are able to find occurrences of possible sensitive data disclosure. While their approach is promising for detecting data disclosure, they, unlike AsyncShock, do not consider multi-threaded code in enclaves.

6 Conclusion

This paper analyses the impact of synchronisation bugs inside SGX enclaves. We have shown that the impact of synchronisation bugs is greater within SGX enclaves than in traditional applications, because their exploitation becomes highly reliable through attacker-controlled scheduling. We described AsyncShock, a tool for thread manipulation, and showed how it can be used to exploit synchronisation bugs by widening the attack window through controlled thread pausing and resuming. AsyncShock operates as a preloaded library without modifications of the target application or host OS. We demonstrated that synchronisation bugs can be exploited inside SGX enclaves using AsyncShock for control flow hijacking or bypassing access checks.

Acknowledgements

We would like to thank the anonymous reviewers for their input. This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 645011 and No. 644412.

References

1. *ARM TrustZone.* `http://www.arm.com/products/processors/technologies/trustzone/index.php`
2. Baumann, A., Peinado, M., Hunt, G.: *Shielding Applications from an Untrusted Cloud with Haven.* In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pp. 267–283 (2014)
3. Bishop, M., Dilger, M.: *Checking for Race Conditions in File Accesses.* Computing systems 2(2), 131–152 (1996)
4. Borisov, N., Johnson, R., Sastry, N., Wagner, D.: *Fixing Races for Fun and Profit: How to Abuse Atime.* In: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, SSYM'05, pp. 20–20 (2005)
5. Checkoway, S., Shacham, H.: *Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface.* In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pp. 253–264 (2013)
6. Chew, L., Lie, D.: *Kivati: Fast Detection and Prevention of Atomicity Violations.* In: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, pp. 307–320 (2010)

7. Dean, D., Hu, A.J.: *Fixing Races for Fun and Profit: How to Use Access(2)*. In: Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, SSYM'04, pp. 14–14 (2004)
8. Flanagan, C., Freund, S.N.: *Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs*. SIGPLAN Not. 39(1), 256–267 (2004)
9. Gao, Q., Zhang, W., Chen, Z., Zheng, M., Qin, F.: *2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs*. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pp. 239–250 (2011)
10. Hariri, Zuckerbraun, Gorenc, *Abusing Silent Mitigations*. BlackHat USA'15
11. Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., Del Cuvillo, J.: *Using Innovative Instructions to Create Trustworthy Software Solutions*. In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP '13, 11:1–11:1 (2013)
12. Intel, *Intel(R) Software Guard Extensions Programming Reference, Revision 2*. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`
13. Intel, *Intel® Software Guard Extensions SDK for Linux* OS , Revision 1.5*. `https://01.org/intel-software-guard-extensions/documentation/intel-sgx-sdk-developer-reference`
14. Jurczyk, M., Coldwind, G.: *Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns*. In: Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns, p. 69 (2013)
15. Lu, S., Park, S., Seo, E., Zhou, Y.: *Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics*. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 329–339 (2008)
16. Lu, S., Tucek, J., Qin, F., Zhou, Y.: *AVIO: Detecting Atomicity Violations via Access Interleaving Invariants*. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, pp. 37–48 (2006)
17. Lucia, B., Ceze, L.: *Finding Concurrency Bugs with Context-aware Communication Graphs*. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 553–563 (2009)
18. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: *Innovative Instructions and Software Model for Isolated Execution*. In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP '13, 10:1–10:1 (2013)
19. PaX, *PaX address space layout randomization (ASLR)*. (2003)
20. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
21. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: *VC3: Trustworthy Data Analytics in the Cloud Using SGX*. In: Security and Privacy (SP), 2015 IEEE Symposium on, pp. 38–54 (2015)
22. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: *AddressSanitizer: A fast address sanity checker*. In: Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 309–318 (2012)

23. Serebryany, K., Iskhodzhanov, T.: *ThreadSanitizer: data race detection in practice.* In: Proceedings of the Workshop on Binary Instrumentation and Applications, pp. 62–71 (2009)
24. Sinha, R., Rajamani, S., Seshia, S., Vaswani, K.: *Moat: Verifying Confidentiality of Enclave Programs.* In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, pp. 1169–1184 (2015)
25. Tsafrir, D., Hertz, T., Wagner, D., Da Silva, D.: *Portably Solving File TOCTTOU Races with Hardness Amplification.* In: FAST'08, pp. 1–18 (2008)
26. Twiz, Sgrakkyu, *Attacking the Core: Kernel Exploitation Notes.* Phrack 64 file 6
27. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: *Ad Hoc Synchronization Considered Harmful.* In: OSDI, pp. 163–176 (2010)
28. Xu, Y., Cui, W., Peinado, M.: *Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems.* In: Security and Privacy (SP), 2015 IEEE Symposium on, pp. 640–656 (2015)
29. Yang, J., Cui, A., Stolfo, S., Sethumadhavan, S.: *Concurrency Attacks.* In: Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism, (2012)
30. Yang, J., Cui, H., Wu, J., Tang, Y., Hu, G.: *Making Parallel Programs Reliable with Stable Multithreading.* Commun. ACM 57(3), 58–69 (2014)
31. Yu, Y., Rodeheffer, T., Chen, W.: *RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking.* In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05, pp. 221–234 (2005)
32. Zhang, W., Sun, C., Lu, S.: *ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach.* In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, pp. 179–192 (2010)

5 Enabling Secure Modularity in Enclaves

AsyncShock has shown that the new threat model for enclaves is different enough that enclave developers need to be aware of previously less important classes of bugs, such as synchronization bugs. Mitigation techniques are one aspect that can help to harden an enclave application; however, only shipping an enclave when the code is 100% hardened is not feasible. Businesses cannot wait until their product is perfect before putting it to market, especially in a fast moving environment such as the cloud. Traditionally, these issues are solved by shipping software updates which can fix bugs, enhance performance and add new features.

Updating software for consumers is usually simple. A new update is installed and, after the software is restarted, it can be used again in its updated state. For businesses that provide, e.g., a Software-as-a-Service (SaaS) offering, the reality looks different. Here, the business critical software, such as database systems, cannot simply be restarted as it would incur downtime. Usually, this is mitigated by announcing maintenance windows with scheduled downtime or by replicating the service so that no downtime is needed if possible. However, if the service is stateful or resources are scarce, downtime or replication might not be possible.

A third approach is to design the application in such a way, that it supports live patching its code while the application is running, a technique known as hot-patching. This also isn't a new technique and is supported by, e.g., big database systems [61, 69]. Even operating systems [39, 25] offer hot-patching to enable kernel updates without downtime.

Similar to hot-patching, there exist a technique to enhance an application with new features so widely used that it's sometimes forgotten that it can be used for this purpose: dynamic loading. Dynamic loading is commonly used to externalize parts of an application. This saves memory during execution as only those parts that are actually needed are loaded. It also makes the parts independently updatable from the main application. Lastly, it can also save space on disk. This is done for libraries that are used by many applications on a system by only shipping the library once and to make sure all applications are using the newest possible version. The dynamic linker is responsible for loading all required libraries at the application start and then link the application and libraries together in memory. An application, however, can also invoke the dynamic linker directly to load code after start up, e.g., to load additional features. This is done by database systems [61, 69], webservers [71] and more.

When stepping into trusted execution, however, neither hot-patching nor dynamic loading were available to developers either due to restrictions of the TEE, such as, e.g., no possibility to include changes into an attestation report, or because no framework existed that provides these features. Neither SGX nor TrustZone were envisioned with large, extensible applications in mind which is a use case that has emerged after launch of SGX when researchers and developers could explore the functionalities of SGX. Only AMD SEV offered a suitable way which allowed large applications to be launched in a TEE but SEV had its own issues that made it impractical to use [24].

5.1 sgx-dl: Dynamic Loading and Hot-Patching for Secure Applications

In *sgx-dl: Dynamic Loading and Hot-Patching for Secure Applications*[97] (on page 67) the sgx-dl framework is presented which provides developers with the ability to enhance their SGX application with the use of dynamic loading and hot-patching. sgx-dl is not the first system that enables dynamic loading in SGX enclaves, however, it is the first system that does it without compromising the security of the enclave.

Existing systems such as SCONE [9], SGXLKL [82] and Graphene-SGX [89] were built on top of SGXv1 which lacks support for changing page permissions during enclave runtime as shown in Section 2.1.2. These early systems aim to load unmodified applications into an SGX enclave and circumvented the SGXv1 restrictions by mapping the entire enclave address space as read-write-executable to enable code loading after enclave initialization, which is a dangerous workaround that enables return-oriented programming and arbitrary code execution attacks. sgx-dl is built on top of SGXv2 and makes use of the SGX SDK to load code at runtime securely and also aims to not load complete unmodified applications but rather enable partitioned application to load dynamic code.

In detail, sgx-dl contains a code leader and a dynamic linker and offers a similar programming interface to `libdl` (`dlopen`, `dlsym`). sgx-dl is programming language agnostic and works by directly operating on Executable and Linking Format (ELF) files. Functions are first collected from unlinked ELF files and registered inside the enclave to be loadable. On call, the function is loaded and linked dynamically and then called. Functions are patched by providing an ELF file with updated code which causes sgx-dl to update the loaded code and relink the function.

To use sgx-dl, enclave developers need to link their enclave against the sgx-dl library. Then, three steps are needed to call a dynamically loaded function. First, the build procedure of the enclave needs to change. Functions that should be loaded later now must be built into their own unlinked ELF object file. This ELF file is later transferred to the enclave when needed so to ensure authenticity and integrity, the file should either be encrypted and signed or transferred over a protected channel.

Second, the enclave code has to register the dynamic functions by calling `dl_add_file` and `dl_add_fcts`. These functions register the file, possible global variables as well as functions and need to be called before any dynamic function can be called. Lastly, all calls to now dynamic function need to be replaced with calls to `dl_begin_call` and `dl_end_call`. These functions facilitate the loading of the dynamic function with `dl_begin_call` returning a function pointer that can be called and `dl_end_call` invalidating that function pointer. Since replacing function calls like this suffers from code-readability issues, sgx-dl also offers another way of calling dynamic functions. Similar to how the SGX SDK generates wrapper functions that represent enclave functions to call from outside the enclave and vice versa, sgx-dl offers a way to generate wrapper functions for dynamic functions. This way, developers can still call dynamic functions like they normally would and let the generated wrapper do the calls to `dl_begin_call` and `dl_end_call`.

Hot-patching a function is achieved by calling `dl_patch` with a reference to the new ELF file and a short patch description stating which functions to update from the given file. sgx-dl will then prevent all calls to dynamic functions that are updated or referenced by other functions, replace the code of the functions-to-patch, relink all loaded functions that reference a replaced function and then allow calls again. Briefly stopping execution is necessary to safely replace the code as the code size might have changed and a new place for the function might need to be found.

To measure the performance impact of having to go through the sgx-dl library to use dynamic functions, a microbenchmark was used that added 10,000 dynamic functions to an enclave and loaded those 1,000 times. Both adding and loading function are operations that are seldom done and were quick with an average time of 229.9 µs and 55.1 µs respectively. Calling a dynamic function also incurs an overhead; however this overhead is not static but rather scales linearly with every dependency a dynamic function has as sgx-dl has to check that all dependencies are also loaded. To measure the calling overhead, we opted to use three macrobenchmarks as those reflect real-world use-cases: (1) STANlite, an SGX-aware database management system built on SQLite which represents an application in which only a small part of the enclave are dynamic functions (the SQL processing engine); (2) LibSEAL, a drop-in replacement for TLS libraries such as OpenSSL/LibreSSL that utilizes application specific auditing modules which represents an application, where without sgx-dl the whole enclave would have to be tailored to each application to include the correct audit module but now with sgx-dl this enclave is generic with the correct audit module being loaded on demand; and (3) *E*Actors, a secure actor framework that utilizes sgx-dl to load the actor code dynamically.

The macrobenchmarks show that the performance overhead from using sgx-dl was below 1.5% in all cases. We also show that using hot-patching the SQL processing engine in the STANLite application was able to cut down application downtime to 4 ms compared to a traditional state-saving to disk, full enclave restart with new code and loading the state again. In conclusion, sgx-dl enables dynamic loading inside SGX SDK enclaves and was the first framework to offer hot-patching support. sgx-dl achieves this while introducing very little performance overhead to the application and offering a secure way of loading code that does not compromises enclave security like other systems before it did.

5.2 Related Work

Dynamic linking is commonplace in current operating systems and its applications when not utilizing trusted execution. It allows developers to decouple common code from their application into shared libraries to reduce application size and make it easier to update said libraries. In theory, shared libraries are also only loaded into memory once and therefore applications can share the memory pages of the library. With today's storage and memory sizes, however, the argument for size reduction and page sharing is outdated [86]. Updating shared libraries, however, is not always easy as the ABI of the shared library might change. A good example is the GNU C standard library (`glibc`), probably one of the most used shared libraries worldwide, which exports its functions with *version tags* [32]. An application linked against an old version of the `glibc` can still use a newer version of it even when the used functions have changed with ABI breakage as long as the new version of the `glibc` still exports the old version with the appropriate version tag. This practice, however, requires manual work and is not done by every shared library.

Some programming languages have reduced their usage of dynamic linking for their applications. Go by default tries to produce statically linked libraries which succeeds when the application does not require some functionality, e.g., name resolution via the `net` package. If it does, go instead creates a dynamically linked binary. The go documentation doesn't actually state this directly, but it's a well known and accepted fact [87]. Nevertheless, go can still be forced to produce statically linked binaries with the right linker flags in such cases. For sgx-dl, however, where dynamic *loading* is used together with dynamic *linking* for loading modules at runtime, this trend has little relevance as using the dynamic linker for modularization is still a valid use case [86].

Compared to dynamic linking, hot-patching, however, is still an active area of research even without trusted execution. Katana [21] is a programming language agnostic system that operates on ELF files and enabled hot-patching a running application. It introduces the *Patch Object* data structure which contains the difference in code and data between two versions. Katana leverages DWARF debugging information to gather type data to allow automatic transformation of changed data structures. The authors, however, concede that this is actually very limited. Automatic patching of data structures is impossible in the general case and the authors advise that "[t]he general solution to all of these type problems is to ask the programmer for routines which perform the application-specific work" [21]. sgx-dl achieves the same goal with different approaches. Instead of a new *Patch Object* data structure, sgx-dl operates on unlinked ELF files plus a patch description which makes it more flexible as the same ELF file with different patch descriptions can be used, depending on the loaded modules which is contrary to Katana which assumes an application that does not load modules at runtime which could be patched. Also, instead of trying to automatically patch data structures, sgx-dl does exactly what Katana proposes and asks the developer to write a transformation function instead of trying to guess how to do it. Lastly, sgx-dl works with the restrictions imposed by utilizing SGX, such as using the SGX mechanism for changing page permissions.

Hot-patching is not only applicable on the application level but also at the kernel level. Systems that can hot-patch a running Linux kernel such as Ksplice [10], kpatch [72] or kGraft [30] are specialized to that task and utilize mechanisms not available in SGX such as stopping all processor cores at will to apply an update in kernel space.

While sgx-dl is the first framework that offers dynamic loading and hot patching for partitioned applications, there have been other related approaches in the past. Microsoft's VC3 [77] was the first system that enabled dynamic loading in a partitioned MapReduce application. It offers modularity by dynamically loading map and reduce functions but no hot-patching support. A different approach is taken by Intel's Protected Code Loader (PCL) [44]. It works together with the SGX SDK and offers a generic loader enclave that in turn loads the actual encrypted enclave code after launch to hide the enclave code. This approach always loads the whole, statically linked enclave at launch and does not offer true dynamic loading at runtime nor does it offer hot-patching. SGX-Shield [80] and SGX-Armor [83] follow the PCL's design and add Address Space Layout Randomization (ASLR) to defend against, e.g., controlled-channel attacks [100]. With ASLR, the location of loaded code is randomized to make it harder for an attack to infer the location of certain code pieces, but this randomization only occurs at load time.

DynSGX [84] comes close to sgx-dl but uses a different architecture: a client-server system that performs remote function calls. Functions are loaded over the network into a remote enclave (the server) and then called from the client. Linking is done completely on the client before transmitting the function to the server. Hot-patching is not stated to be supported, but could be emulated by unloading a function and loading a new version of it. With DynSGX, the enclave cannot execute on its own and the calling application has to make network round-trips to the remote enclave to call its functions.

Other systems [55, 20] also exist, but they are variants that don't support dynamic linking. Instead, they just copy a statically linked blob into an enclave and can jump to it. Hot-patching is not directly supported but can, again, be emulated by unloading one blob and loading another.

(Publication starting next page)

Experience Paper: sgx-dl: Dynamic Loading and Hot-Patching for Secure Applications

Nico Weichbrodt
weichbr@ibr.cs.tu-bs.de
TU Braunschweig
Germany

Joshua Heinemann
heineman@ibr.cs.tu-bs.de
TU Braunschweig
Germany

Lennart Almstedt
almstedt@ibr.cs.tu-bs.de
TU Braunschweig
Germany

Pierre-Louis Aublin
pierrelouis@iij-ii.co.jp
IIJ Innovation Institute
Japan

Rüdiger Kapitza
kapitza@ibr.cs.tu-bs.de
TU Braunschweig
Germany

ABSTRACT

Trusted execution as offered by Intel's Software Guard Extensions (SGX) is considered as an enabler to protect the integrity and confidentiality of stateful workloads such as key-value stores and databases in untrusted environments. These systems are typically long running and require extension mechanisms built on top of dynamic loading as well as hot-patching to avoid downtimes and apply security updates faster. However, such essential mechanisms are currently neglected or even missing in combination with trusted execution.

We present sgx-dl, a lean framework that enables dynamic loading of enclave code at the function level and hot-patching of dynamically loaded code. Additionally, sgx-dl is the first framework to utilize the new SGX version 2 features and also provides a versioning mechanism for dynamically loaded code. Our evaluation shows that sgx-dl introduces a performance overhead of less than 5% and shrinks application downtime by an order of magnitude in the case of a database system.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; • **Security and privacy** → *Software security engineering*.

KEYWORDS

Intel Software Guard Extensions, Trusted Execution, Dynamic Code Loading, Hot-Patching

ACM Reference Format:
Nico Weichbrodt, Joshua Heinemann, Lennart Almstedt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2021. Experience Paper: sgx-dl: Dynamic Loading and Hot-Patching for Secure Applications. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Québec city, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3476134

1 INTRODUCTION

Trusted execution is considered as a solution to improve the ruptured reputation of cloud infrastructures [20, 23, 52] as it partly shifts the needed trust from the cloud infrastructure and its provider to the hardware and its manufacturer. It can protect the integrity and confidentiality of applications even against privileged software. To this day, Intel SGX, which is the most studied Trusted Execution Environment (TEE), has been adopted by various cloud vendors such as Alibaba cloud [17], Microsoft Azure [1] or IBM [28].

Trusted Execution benefits applications that store and process confidential data such as databases [10, 46, 48] or web servers [6, 61]. These complex applications are usually dynamically linked and often feature extension mechanisms to load additional code on demand. This reduces start-up costs as initially only the core functionality of an application needs to be loaded, but also paves the way for hot-patching. Fast (re-)starts are important as users of cloud applications expect them to always be available, despite the need for updates to add new features, fix bugs and address security vulnerabilities. The latter especially applies for stateful services that serve multiple tenants where downtimes due to restarts are difficult to coordinate and provisioning of another copy of the service would require a costly state transfer. Hot-patching can prevent downtime and performance disruption due to a cold restart, enables faster bug and security vulnerabilities fixes, and makes new features available sooner. Especially the faster deployment of security critical bug fixes has made hot-patching a key feature for databases and virtual machines. In distributed systems, hot-patching can prevent the degradation of a high-availability setup or cluster when one replica has to be restarted. Major commercial softwares such as Windows Server VMs on Azure [2], Canonical Livepatch [4], Azure SQL Database [37] and Oracle Database 11g [41] support hot-patching. Even more, it is intensively used in the cloud, with millions of SQL servers hot-patched every month [3].

Unfortunately, the support for dynamic loading and hot-patching in TEEs, and especially Intel SGX, is limited. Partly this was caused by restrictions of the first version of SGX. To protect an application with SGX, an *enclave*, resembling a trusted execution context, has to be instantiated. With the first version of SGX, enclave initialization required a sequential page-wise loading of the binary with the enclave page layout being immutable after creation. This property prevents any extensibility at runtime, including security patches. So far a number of systems [6, 33, 61] circumvented these restrictions by mapping the entire enclave address space as read-write-executable to enable code loading after enclave initialization, which is a dangerous workaround that enables return-oriented programming and arbitrary code execution attacks.

This leaves an entire application restart as the only security-sensible option to add new functionality and to patch vulnerabilities. For large enclaves, required for stateful applications, this results in a prohibitively long start-up time, due to size restrictions of the protected memory. To illustrate this problem, we have extended the STANlite [48] secure database with support for state transfer and have measured the throughput over time in an experiment where a new, updated enclave is created and state transfer is executed. In our evaluation (see §5.5), we show that such a restart with a state that fits into protected memory causes a downtime of ≈ 1.20 s in a best-case scenario. Even with an efficient state transfer between the old and new instance of the application, the restart can be quite costly and take much longer for larger application states. On top of that, it is necessary to prevent rollback attacks on the saved state via a suitable mechanism [11, 35, 57], which further increases the full restart duration. The alternative of starting a new instance in the background to hide the start-up process is also not suitable as this would double resource demand as now two enclaves are running side-by-side.

In essence, to match the functionality and performance of unsecured stateful cloud applications while also preserving security, a new foundation and framework are necessary. With SGX version two (SGXv2), a foundation now exists as it offers support for sensible memory-access management that can be used to devise dynamic loading and hot-patching support. In this paper we propose sgx-dl, a novel framework for dynamic code loading and the first framework supporting hot-patching in Intel SGX enclaves. In particular, sgx-dl makes the following three contributions:

(1) sgx-dl enables dynamic loading based on the plain support of the Intel SGX Software Development Kit (SDK) and it's SGXv2 capabilities while being minimally invasive. sgx-dl thereby implements a minimal-Trusted Computing Base (TCB)-centric and efficient dynamic loading approach while preserving $W \oplus X$ semantics at all times.
(2) sgx-dl features function-level hot-patching to apply updates at runtime without requiring a restart. Furthermore, compared to other hot-patching systems, sgx-dl actively removes outdated and stale code and data to increase security.
(3) sgx-dl introduces a novel versioning mechanism that enables version reporting of the dynamic enclave code. This enables enforcement of a specific version of dynamic code even after updates, which previous dynamic loading systems for SGX did not provide.

To evaluate sgx-dl, three applications have been extended: the STANLite [48] in-enclave database; the LibSEAL [8] auditing library and the *E*Actors [47] trusted actor framework. Our evaluation shows that applications utilizing sgx-dl can easily benefit from dynamic loading of code while impacting their performance by less then 5% and reducing downtime when patching to about 4 ms.

The remainder of the paper is organized as follows: First, in §2, we briefly introduce Intel SGX and detail the current state of the art regarding dynamic loading of functions in traditional systems and inside SGX enclaves. In §3, we present the design of sgx-dl with details on its interface in §4. Lastly, we evaluate sgx-dl in micro- and macrobenchmarks in §5 and conclude the paper in §6.

2 BACKGROUND AND RELATED WORK

This section is subdivided into five parts. First, we give an overview about hot patching in commodity systems. Second, a compact primer about SGX is provided. Third, we show the evolution of dynamic loading in SGX systems. Fourth, we describe our assumed threat model and lastly, we present existing dynamic loading systems and their shortcomings.

2.1 Dynamic Loading and Hot-Patching in Traditional Systems

Most of today's applications are *dynamically linked*: They do not link their dependencies into the binary but instead reference them as individual libraries that are installed on the system. This allows sharing of libraries between multiple applications which keeps application binaries small and allows updating libraries independently from applications using them.

On application start, the *dynamic linker* is responsible for resolving the dependencies by loading the libraries and inspecting the relocation sections inside the application binary (on Linux systems inside the Executable and Linking Format (ELF) file) to figure out which symbols need to be linked to which libraries, at which position inside the application's code this linking should take place, and what type of linkage it is. Additionally, with library functions like `dlopen`, the application can decide during runtime to load a specific dynamic library. This is usually done to enable modularity.

When a library gets updated, the application has to be restarted to pull in the new code during the next linking phase at relaunch. However, restarting the whole application might not be desirable due to the cost of restarting it: the application might need to save its in-memory state to disk first, which then has to be reloaded when starting again and cannot reply to requests during that time.

Therefore, systems exist to update code during application runtime without restarting the application, a practice known as *hot-* or *live-patching*. Examples of such systems are the language-agnostic Katana [13] for ELF binaries, Kitsune [22] for applications written in C, and Pymoult [34] for Python applications. Languages that run on top of a virtual machine, e.g., Java, can often utilize the code loading functionality of the virtual machine to implement hot-patching. Hot-patching can also be performed on the underlying operating system, e.g., with Ksplice [7], a system that allows patching parts of the Linux kernel without rebooting it. Open source systems with similar approaches are kpatch [45] and kGraft [18] which differ in the way they ensure runtime consistency of the updated code. The ideas of all these systems are not directly applicable to SGX as they are either language dependent or use mechanisms not available in SGX, however, they can serve as a basis for a hot-patching system.

2.2 Intel Software Guard Extensions (SGX)

At the end of 2015, Intel released processors of their Skylake microarchitecture which was the first to incorporate support for the Intel Software Guard Extensions (SGX). SGX allows the creation of so-called *enclaves* which are parts of normal user-space applications that are executed in a secure execution mode and reside in a special memory area called the Enclave Page Cache (EPC). This enables SGX to act as a TEE as the new execution mode and EPC guarantee that i) enclave data is always encrypted and integrity protected inside the EPC, ii) each enclave can only access its own data and non-enclave data in unencrypted form and iii) enclave execution is isolated and hidden from other processes and privileged software such as the operating system. The EPC is part of the Processor Reserved Memory (PRM) that also hosts other SGX-specific data structures. On current[1] commercially available SGXv1 systems, the PRM typically has a size of 128 MiB with the EPC portion being 93 MiB big while SGXv2 machines can go up to 256 MiB PRM. Nevertheless, this is more than an order of magnitude less than the amount of available main memory and will for larger enclaves still result in costly enclave paging which can decrease enclave performance drastically [6, 14].

Intel SGX offers a new set of instructions to create and manage enclaves and multiple approaches to develop enclaves have emerged during the last years. Intel's own SGX SDK [26] can be used to either integrate an enclave into an existing application, assuming one has access to the source code, or to build new applications from scratch. These SDK applications are commonly known as *partitioned applications*. Another approach is to use third-party SDKs [21, 66] or to embed additional code inside the enclave, up to a library OS, to offer standard functionalities to applications executing inside it [10, 33, 61]. In this paper, we focused our work on the official Intel SDK because it is the most common way to secure applications via SGX but the design can be adapted to other systems as well.

2.3 Towards Dynamic Code Loading

Library OS-based systems are able to host an unmodified application inside an SGX enclave. They need to be able to dynamically link this application inside the enclave which requires the ability to modify and load code. Normally, this is not a problem as code pages can be mapped as read-writeable to be modified and then remapped again as read-executable. However, this raises an important issue for library OSs running on SGX: after creating an enclave, its memory layout and page permissions cannot be changed. Therefore, these approaches have to map the entire enclave as read-write-executable, which makes them easier to exploit [9].

The authors of Haven [10] therefore proposed an extension of SGX's feature set, namely the possibility to add/remove pages after enclave creation as well as changing permissions of enclave pages. Intel accepted this idea and developed SGX version two (SGXv2). At the start of 2018, the first processors[2] with SGXv2 became available. Support for SGXv2 also arrived in the SGX SDK with version 2.0 in November 2017, right when first SGXv2 processors were launched. With SDK version 2.8, support for an enclave-managed and dynamic memory region called *reserved memory* was added. Reserved memory is neither part of enclave heap nor stack but has to be managed by enclave code, sort of like `mmap`ed memory. Permissions of reserved memory are freely changeable on a page granularity, with a call similar to `mprotect`.

[1]As of December 2020

[2]Intel Gemini Lake; Ice Lake-U in 2019

Being able to dynamically load and execute code not only demands for dynamically adding pages to enclave memory, but also for a dynamic linker. Inside an SGX enclave, no dynamic linker is normally available. SGX-LKL [33] and Graphene-SGX [61] do contain one as part of the whole library OS, which is used for the initial loading of the application. An update, however, of the application or its libraries with re-linking is not possible without restarting the whole enclave with those systems. The same is true when developing an application based on the SGX SDK as a new enclave that contains the updated code has to be launched.

Both library OS and partitioned applications have their advantages and disadvantages, e.g., the TCB of a partitioned application is typically smaller compared to a library OS approach whereas the library OS requires no or only slight modifications to the application it should run. Contrary to the presented approaches, sgx-dl focuses on a as small as possible codebase to reduce the number of possible bugs while offering dynamic loading and hot-patching.

2.4 Threat Model

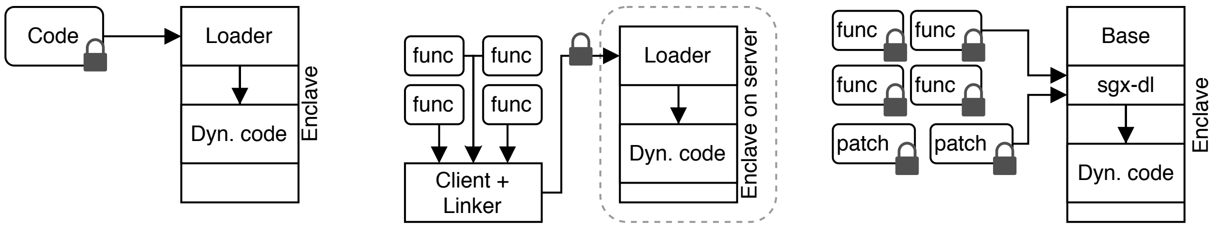
In this paper, we assume the following threat model:

We assume that SGX and the cryptography it uses are not broken, i.e., during enclave execution an attacker is not able to tamper with the enclave except calling the enclave's public interface. We assume that an attacker has privileged access to the machine hosting the enclave. Denial-of-Service is not considered a valid attack as it is always possible to just refuse to execute the enclave. Microarchitectural side channel attacks against SGX [16, 30, 32, 39, 40, 50, 62, 63, 65] are not considered, as there have been microcode [25] and SDK [27] updates in the past or they will be fixed in the future. Application side-channels such as controlled-channel attacks [64, 67, 68] or others are also not considered. Mitigations such as Address Space Layout Randomization [12, 51, 53], oblivious memory accesses [5, 42, 43], constant time programming [15, 56] and others [54] exist.

2.5 Dynamic Loading in Partitioned Systems

When looking at dynamic loading in SGX, there exist systems that support it that are not a library OS. Microsoft's *VC3* [49] is a secure MapReduce [19] framework and uses dynamic loading to load encrypted map and reduce functions. However, no generic code can be loaded, just map and reduce functions. In 2017 Intel released the Protected Code Loader (PCL) [24] library. It can be used in combination with the SGX SDK to create generic loader enclaves that can load encrypted enclave code. At enclave start, the encrypted code is loaded and decrypted by the loader code. The encrypted code itself is statically linked, no single functions can be loaded, only whole enclaves, and the enclave can only contain the PCL loader code, see Figure 1a. Updates at runtime are not supported. This design is also utilized by other systems, such as *SGX-Shield* [51] and *SGX-Armor* [53]. They add Address Space Layout Randomization (ASLR) of the loaded code to defend against, e.g., controlled-channel attacks [68]. Enclave code is randomized when loading the enclave such that an attacker does not know where code is located.

DynSGX [55] is a framework for dynamically loading encrypted code into enclaves regardless of their functionality. It utilizes a client-server model: a generic DynSGX enclave is loaded and a



(a) statically linked and encrypted code loading on the same machine

(b) function loading with linking on separate a client

(c) function loading, linking and hot-patching on the same machine

Figure 1: Different approaches to dynamic loading: (a) Intel PCL, SGX-Shield, (b) DynSGX, and (c) sgx-dl

secure communication channel is established from the secure client to the enclave. This hinders its practicality and versatility as the enclave cannot execute on its own and additional code is required on the client to be able to communicate with a DynSGX enclave. Code is sent to the enclave from the client and executed there on demand, similar to a remote procedure call, see Figure 1b. The client is doing the dynamic linking on its machine and only sends linked executable code to the enclave. Hot-patching is not directly supported, but the client could update its unlinked code before linking and sending it to the enclave.

Other systems [12, 31] also exist but they are variants that essentially load statically linked code.

In summary, all of these systems allow dynamic loading of code with varying degree of additional features. None of the systems, however, support hot-patching the dynamic code after it has been loaded. With sgx-dl, we propose a novel approach as shown in Figure 1c by including a linker inside the partitioned enclave to enable function-level dynamic loading and hot-patching. Compared to DynSGX, a client is not involved in the linkage and does not need to handle server communication differently. Furthermore, enclaves using sgx-dl can also include common *base* code that is always present inside the enclave.

3 THE DYNAMIC CODE LOADER SGX-DL

We outline the following requirements a dynamic code loading system for SGX should fulfill:

R_1 Load and link possibly encrypted generic code.
R_2 Allow hot-patching of code without an enclave restart.
R_3 Offer a versioning mechanism that reflects the state of the loaded code.

In the following we will show how the different requirements are fulfilled by sgx-dl.

3.1 Loading and Linking Encrypted Code

The core of sgx-dl consists of the dynamic loading and linking of functions and data in the most flexible way: Instead of only being able to load whole binary blobs, loading is done on a function level. This way, the developer can decide very fine-grained how to extend an application which increases modularity. We call these functions loaded at runtime *dynamic functions*. Such dynamic functions have to be loaded to a location that allows writing code to it. The SGX SDK provides such a location with the *reserved memory* which is used by sgx-dl to store all dynamic functions and data. Loading code and data is not enough, it also has to be linked. Therefore, a linker is part of sgx-dl.

Dynamic functions that should be loaded need to come from a trusted source in some kind of container, e.g., an ELF file. This can be achieved by compiling, but not linking, the dynamic functions on the enclave developer's machine. Then, the enclave can retrieve the dynamic functions either by reading them from a local encrypted file or over an encrypted network connection. In both cases the enclave has to decrypt the code and perform integrity checks, such as checking a Message Authentication Code (MAC) or a signature. These mechanisms allow loading of generic code and linking it to other loaded code, but it is not enough to fulfill requirement R_1 yet as shown in §3.2.

3.2 Handling of Enclave Functions

With sgx-dl, dynamic functions should be able to call functions that were present at enclave launch, e.g., functions of the SDK, standard library or of the application itself. To achieve this, sgx-dl has to parse the symbol table from the ELF file of the enclave.

However, when the enclave is loaded, its symbol table is not present in memory as it is not added to the enclave by Intel's SDK. A potential reason for this is to save enclave memory, as the symbol table is not needed for normal enclave execution. The SDK could be changed to support this, however, we refrain from doing so as i) not all applications require dynamic loading and ii) not changing the SDK maintains sgx-dl's independence of it. We propose a different way to obtain the symbol table: The enclave can open its own enclave file in untrusted memory and copy the symbol table into the enclave to parse it. This poses a security challenge, as it must verify that it loaded the correct symbol table belonging to the current running enclave, which is solved as follows: When building and signing the enclave, the enclave developer extracts the symbol table using a tool like `readelf` and calculates the SHA256 hashsum of it[3] which we call the *Symbol Table Hash (SThash)*. The SThash can then be embedded into the enclave and is passed to sgx-dl on enclave application start. sgx-dl recalculates the SThash and compares it against the provided one. This ensures that only

[3]For example: `readelf -x .symtab enclave.signed.so | tail -n +3 | head -n -1 | awk '{print $2$3$4$5}' | tr -d "\n" | xxd -r -p | sha256sum`

the correct symbol table is loaded. Embedding the SThash does not change the symbol table but requires a second compilation step. The enclave first needs to be compiled with a placeholder value that is then replaced with the SThash before compiling and signing again. This is only needed for the enclave file itself, for dynamically loaded code the ELF is protected as stated in §3.1. With this mechanism, requirement R_1 is now fulfilled.

3.3 Updating Dynamically Loaded Functions

As required by R_2, it should be possible to update dynamically loaded code at runtime. For this, the developer supplies a patch description and an ELF file containing the actual updated code to sgx-dl. These can be obtained via a secure network connection or by loading a file. To authenticate a patch, the enclave can check the patch's signature, or, if a secure channel is used, the patch is implicitly authenticated by the channel. The patch description lists the functions that should be added to, as well as updated in, the enclave. Removal is done automatically as sgx-dl removes all functions and data that are no longer needed, i.e., not linked against, from the enclave after applying the patch. Applying a patch will stop execution of dynamic functions briefly to ensure safe removal of the old code. Note that updating a function will cause the old code to be deleted from the enclave. Applying two patches that update the same function will leave no trace of the first patch inside the enclave. With the addition of hot-patching, sgx-dl now fulfills the requirement R_2.

3.4 Versioning of Dynamically Loaded Functions

Intel SGX offers an attestation mechanism to verify that the correct enclave code has been launched. When the enclave is created, a rolling SHA256 hashsum over all pages added to the enclave is calculated. This hashsum is called the *measurement* of the enclave and is compared against an expected value at launch. Using the measurement, a remote party can verify the identity of the enclave.

The measurement, however, can not change after enclave creation. It only represents the state at enclave launch. In case of an enclave that dynamically loads code, this makes it impossible to use the measurement to verify which code has been added. We, therefore, introduce a mechanism into sgx-dl to generate a deterministic hashsum of the dynamically added code and data, called the *Dynamic Code Hash (DChash)*. The DChash serves as a version indicator of the currently registered code and data.

To ensure a deterministic DChash of the dynamic content that is independent of the order in which symbols have been registered, we propose the following mechanism. The library starts in an empty state which will always generate the same DChash that is equal to the SHA256 hashsum of a 0-byte input. After functions have been added to sgx-dl, the library state changes, containing all added functions and objects. When creating the DChash, sgx-dl will sort functions based on the SHA256 hashsum of their code and their symbol name. The preliminary DChash then consists of the code hashsum and symbol name of every function.

For data, the same approach is taken, except that sgx-dl differentiates between read-only and writeable data. Both are sorted by their symbol name which contributes to the DChash. For read-only data, such as string literals or constants, we also include a SHA256 hashsum of their contents in the DChash. Non-constant values cannot contribute a hashsum of their content as execution might change their value and the DChash is intended to reflect the registered code and data and not the internal application state. The final DChash then can be used as a version indicator of the registered code and data. As this value has been generated by sgx-dl inside the enclave, it can be signed by any in-enclave key whose trustworthiness is tied to the attestation report of the enclave. Should the library state change, e.g., because a patch changes a symbol, the DChash will be recalculated.

This system also ensures that the DChash is not dependent on the order patches are applied, but on their content. With, e.g., three patches *A*, *B* and *C* that all change the same symbol, A being the oldest and C being the newest patch, it does not matter if A is applied before B (or even at all) as long as C is last because the content of C is the newest one. Similarly, if those patches would all change different symbols, than the order is completely irrelevant. In a distributed system, this property can be used to ensure that all replicas are on the same version. The presented versioning mechanism integrated into sgx-dl fulfills requirement R_3.

4 SGX-DL IMPLEMENTATION

In this section, we present how the dynamic loader and hot-patcher of sgx-dl is implemented.

4.1 Interface of sgx-dl

sgx-dl is designed as two static libraries that are added to the enclave and to the untrusted application respectively. We opted to implement sgx-dl on top of the SGX SDK but it's techniques and ideas can also be implemented into library OS systems. The trusted enclave part contains an ELF-parser, a linker, a function and object registry, as well as the public interface that developers can use. The interface for code loading is only available inside the enclave and resembles an in-enclave `dlopen`/`dladdr`. The untrusted parts contains methods for opening the enclave file in memory for the trusted part to copy the symbol table so that enclave functions can be discovered. Figure 2 shows on the left side how the trusted library is integrated into the enclave build process and which files will be generated form the source files.

Functions that should be dynamically loaded need to be registered (*added*) with the trusted library as dynamic functions. This creates the necessary management data structures needed for loading the function later and is only needed once per function. The ELF parser is responsible for parsing ELF files to extract functions and relocation tables from it. Extracted functions and objects are registered in the registry. The linker takes the extracted functions and relocation tables and resolves all symbols.

Similar to Ksplice, sgx-dl operates on the ELF file directly and is not inspecting any source code. As ELF files do not contain any information about the functions they contain other than their symbol name, sgx-dl cannot infer the signature of functions or their programming language. Fortunately, this allows sgx-dl to not be restricted to C and C++ code. For example, sgx-dl is able to load Rust [36] functions, provided they were compiled with

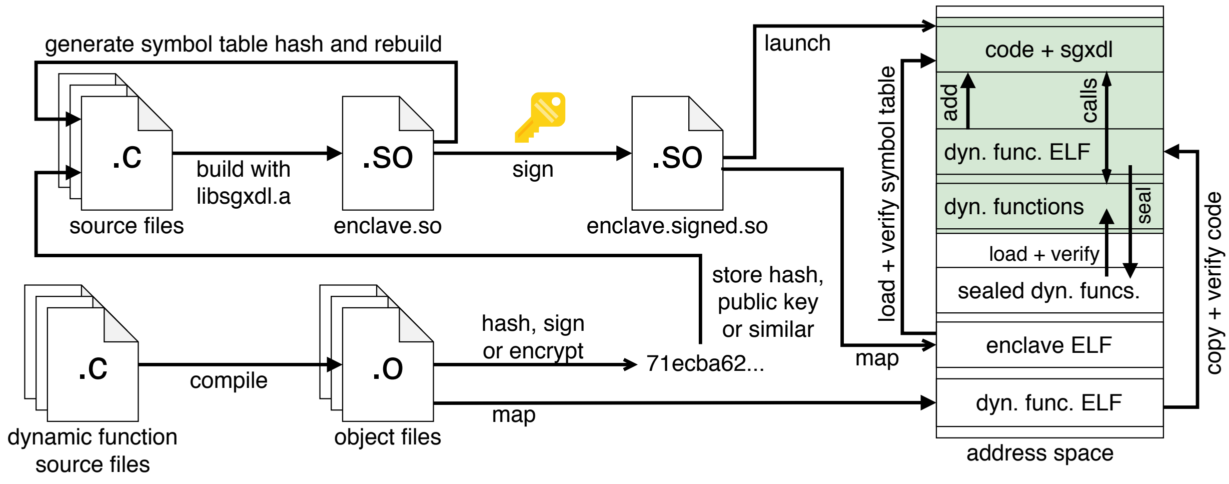


Figure 2: Overview of sgx-dl. On the left, the different files used and generated by the enclave build process are shown. On the right, the memory layout of the enclave inside a process can be seen and how it corresponds to the generated files. The green shaded area is enclave memory.

`dl_add_file`	Add a file and get back a file handle
`dl_add_fcts`	Add functions from a file handle
`dl_begin_call`	Prepare a function for calling and return a pointer to it
`dl_end_call`	Mark a function as done calling
`dl_patch`	Update already present functions

Table 1: API of sgx-dl

`[no_std]` [58] as our enclaves do not contain a Rust standard library. However, one could integrate sgx-dl into one of the available Rust SGX SDKs [21, 66].

4.2 Adding and Loading a Function

To be able to call a dynamic function, it first has to be added and loaded, similar to calling `dlopen` in a non-SGX application. However, sgx-dl is not adding all functions of a library at once but requires the developer to specify which functions to add to keep the enclave code at the minimum and to achieve a minimal TCB. *Adding* a function registers it with sgx-dl, creates the necessary management structures and discovers dependencies whereas *loading* copies the function code, makes it executable and does the linking. Both of those are depicted in Figure 2 on the right side with the separate steps explained in the following sections. The API is shown in Table 1 and also explained in the following sections.

Adding a function. Adding a dynamic function is done using the `dl_add_fcts` method inside the enclave. The developer specifies an array of symbol names to be loaded and provides a file handle for each symbol name that represents a pointer to the ELF file containing the symbol. The file handle is obtained by calling `dl_add_file`. It is expected that the pointer to the ELF file points to in-enclave memory and has been checked for integrity. The library requires the underlying application to verify that the correct ELF file is provided. Integrity verification can be achieved, e.g., by checking hashes embedded inside the enclave or verifying signatures of the loaded ELF file. The ELF file could also have been encrypted or received over the network.

When calling `dl_add_fcts`, for each symbol, sgx-dl extracts the machine code and seals it outside of the enclave with only a hashsum of the code being held inside the enclave in the functions management structure. This hashsum is verified when the code is loaded. While the sealing mechanism of SGX already has integrity protection, we use this additional hashsum as rollback protection. The code is sealed outside to save on EPC usage when a function is not loaded. Note that we seal not to disk, but to untrusted memory.

Next, sgx-dl extracts the relocation table of the symbol. The relocations have to be resolved so that during execution all references to other functions and objects can be linked to their corresponding addresses. This is done by looking at all added functions and objects: sgx-dl iterates through all known functions and their relocation tables. Using this information, sgx-dl builds dependency lists for all dynamic functions by recording all caller functions.

Whenever an ELF file is added, all objects, e.g., global variables, contained in the file are also registered and loaded into the enclave. These objects are always present inside the enclave and are initialized with their respective values. After all dynamic functions from the ELF file have been added, it can be removed from the enclave.

Loading a function. Loading a dynamic function consists of three steps: i) unsealing the code into free memory, ii) loading all dependencies, and iii) replacing all relocations. Dynamic functions are loaded automatically when they or any of their parent functions are called.

The actual code is unsealed to the reserved memory area provided by the SGX SDK. As this area is not managed by the SDK heap memory allocator, we use our own memory allocator to fit dynamic functions into these pages. After the code has been unsealed, sgx-dl compares the hashsum of the unsealed code with the saved hashsum. In case of a mismatch, the code is removed and an error is returned. Otherwise the code has been loaded successfully and dependencies will be loaded. As every function has a list of all its direct dependencies (functions called by itself) as well as indirect dependencies (functions called by functions called by itself) sgx-dl can simply load all dynamic functions from that list to load the full dependency graph. Afterwards, all relocations are linked to addresses pointing to the correct functions and objects making the dynamic function callable. Note that sgx-dl ensures that the reserved memory area is always either read-writable or read-executable but never writable and executable at the same time, thereby preserving $W \oplus X$ semantics.

4.3 Calling a Function

To call a dynamic function, the enclave developer has two options: manual and transparent. The manual option is to call into sgx-dl directly using `dl_begin_call`. This method just requires the name of the function and returns a function pointer to that function. That function pointer is valid until a call to `dl_end_call` and must never be stored by the application after the corresponding call to `dl_end_call`. The transparent option is calling the dynamic function directly like any other function. However, this would normally result in an undefined symbol, which is why sgx-dl needs to generate wrapper code that defines the symbol. The wrapper code internally uses `dl_begin_call` and `dl_end_call` to call to the function. This mechanism is similar to the sgx-edger8r of the SGX SDK that generates the wrapper code for the enclave interface. To generate the wrapper code, sgx-dl needs a list of all functions with their signature which it then turns into a C source and a C header file. The source file is then compiled and linked into the enclave application.

Calling a dynamic function first makes sure that it and all its dependencies are loaded. If the function or one of its dependencies is not loaded, they are loaded automatically in this step. The dynamic function and all its dependencies are marked to prevent them from being unloaded during execution. Then, the dynamic function is invoked and a possible return value is saved. Lastly, the dynamic function and all its dependencies get their marks removed.

The first call to any dynamic function after a new function has been added to sgx-dl causes an internal clean-up of all registered functions and objects. Unreferenced functions and objects might appear after an ELF file was added but not all of its contents were added, e.g., when only one ELF file is used for all dynamic code but not always all code is needed. All unreferenced functions and objects are unregistered to ensure that creating the DChash always represents all the reachable functions and accessible data and does not include any functions or data that is unreachable. The overhead for the first call is therefore increased compared to subsequent calls but this increases security by removing unused code.

sgx-dl supports multithreaded execution of dynamic functions. During execution of dynamic functions, the library is in a read-only state which only allows other threads to execute dynamic functions. Only when no execution is in progress, hot patching, adding and loading functions is done.

4.4 Hot-Patching a Function

Hot-patching a dynamic function is also supported and can be summarized as unloading, replacing and re-loading a dynamic function. In detail, updating works as follows: First, the developer provides a patch description to sgx-dl as described in §3.3. Together with a pointer to the new ELF, `dl_patch` is called. Then, all functions that should be newly added are added as described in §4.2. For all functions that should be updated, the given ELF binary is parsed in the same fashion as adding a dynamic function with the same information being extracted.

Next, the dynamic functions are marked for updating, similar to the execution mark. This is done to ensure that other threads trying to call them or other dynamic functions that depend on them are stopped right before execution. The updating thread then waits for all threads to finish execution in dynamic functions before unloading the functions. The old sealed machine code residing in untrusted memory is then replaced with the new version.

Lastly, the update marks are removed and execution can continue. Lastly, a clean-up routine removes all unneeded functions from the enclave to create a deterministic state for versioning, see §3.4. The next call to the updated function will then automatically load the updated version.

Compared with the existing Ksplice, this differs in a few key points: Ksplice retains the old code and loads the new code to another location. It then inserts a jump to the new code at the start of the old code. sgx-dl does not retain a copy of the old code to save enclave memory and to not increase the TCB with old code. Therefore, sgx-dl also has to relink all dynamic code instead of just changing a jump target.

4.5 Using sgx-dl in an Application

Using sgx-dl in an enclave is simple: the enclave developer builds the enclave as usual and additionally statically links our library into it, see Figure 2. All dynamic functions are then compiled separately into unlinked object files. If code confidentiality is required, these object files can be encrypted. Otherwise, signing or just hashing them is sufficient. When enclave functions should be called from dynamic functions, the SThash of the enclave needs to be embedded into it (see §3.2).

Inside the enclave, the dynamic functions need to be added before they can be called. A minimal code example is shown in Listing 1. In this example, a pointer to the ELF file containing the dynamic functions is returned by the custom `load_and_verify_elf` function which obtains the ELF, loads it into enclave memory and verifies its integrity. The file is added to the dynamic linker in line 6. Then, the functions `foo` and `bar` from this file are added in line 8. Afterwards, `foo` is called using `dl_begin_call` and `dl_end_call` in line 13. `bar` is called in line 16 using the generated wrapper code. Note that `bar` calling `foo` does not need to use sgx-dl API as calls inside dynamic functions can be resolved at runtime.

To hot-patch a function, a pointer to the new ELF file has to be provided as well as a patch description to the `dl_patch` method as shown in Listing 2. sgx-dl will add and update functions from the new ELF file and remove old ones according to the patch description. Afterwards, the updated functions can be called as before.

```c
// dlwrapper.h -- generated wrapper code header
#define foo(a, b) __dynamic_bar(a, b)
int __dynamic_foo(int a, int b);
#define bar(a, b) __dynamic_bar(a, b)
int __dynamic_bar(int a, int b);
```

```c
// code.c -- dynamically loaded code
// Function to load
int foo(int a, int b) {
    return (a + b) * 2;
}

int bar(int a, int b) {
    // No sgx-dl API calls necessary here!
    return foo(a * 5, b / 2);
}
```

```c
// enclave.c -- static enclave code
// elf points to the ELF file containing the dyn functions
#include "dlwrapper.h"
void *elf = load_and_validate_elf();

// Add the ELF file
sgx_dl_file_hndl_t *file = dl_add_file(elf);

// Register (add) functions, order does not matter
dl_add_fct("foo", file);
dl_add_fct("bar", file);

int returnvalue = 0;
// Call the function like this
int (*p_foo)(int, int) = NULL;
dl_begin_call("foo", &p_foo);
returnvalue = p_foo(21, 42);
dl_end_call("foo");

// Or like this
returnvalue = bar(21, 42);
```

Listing 1: Minimal example for using sgx-dl.

```c
// update.c -- patch code
int bar(int a, int b) {
    return (a - b) * 4;
}
```

```c
// enclave.c -- static enclave code
dl_patch_desc_t my_patch = {
    .add_symbols = NULL,
    .update_symbols = "bar"
};

void *elf = load_and_validate_elf();
// for pachting no file handle is needed
dl_patch(elf, my_patch);
```

Listing 2: Hot patching code with sgx-dl

4.6 Restrictions for Dynamic Functions

As with other hot-patching systems, sgx-dl is not able to patch code that is currently executing. This entails that no long-running dynamic functions should exist to ensure an update is eventually applied. The dynamic functions need to return at some point back to the base enclave such that patching can commence. One way to solve this is to implement an explicit signaling mechanism, e.g., a global variable which is periodically checked by the executing dynamic functions. This variable can, e.g., be set when receiving a specific request, to a specific value indicating the dynamic functions should exit.

When updating functions, the developer has to be aware of symbol-name conflicts and dependencies to previously loaded functions and objects. The provided ELF file should not contain redefinitions of already loaded objects (e.g., global variables) as they will be added and loaded again as a second independent copy. This new variable is then only referenced by the updated function.

This mechanism can be used to change the size of objects by updating all functions accessing the object and redefining it with the new size. A state transfer from an old object to a new object can be achieved by executing the following steps: (1) add a function S that serializes the object state into a given buffer; (2) call S and give it a buffer of sufficient size; (3) update all functions accessing the object to new versions; (4) add a deserialization function D that deserializes and converts the old state into the new object from a given buffer; and (5) call D with the buffer from step (2). After this, the enclave now has successfully completed a state transfer for the object. S and D can now be removed, as they are no longer needed.

5 EVALUATION

To evaluate the performance impact of sgx-dl, we design two sets of benchmarks: i) A microbenchmark to show the performance cost of adding and loading functions, and ii) benchmarks in which we integrate sgx-dl into different, SGX-enabled applications to show the real-world performance impact of calling dynamically loadable functions as well as to evaluate the impact of hot-patching a dynamically loaded function. Furthermore, we discuss the security implications of using sgx-dl.

5.1 Test Environment

sgx-dl requires an Intel SGXv2-capable processor. To our knowledge, SGXv2 is only available in Intel's Gemini Lake and Ice Lake-U series of processors and currently no cloud provider offers machines with SGXv2 capabilities. Our SGXv2 test machine is a Dell XPS 13 7390 laptop with an Intel Core i7-1065G7 quad-core CPU (1.3 GHz fixed frequency) and 16 GB of dual-channel LPDDR4-3773 memory. This machine has 188 MiB EPC memory available for enclaves. We use Ubuntu Linux 18.04.4 with kernel version 4.15.0-88. The microcode package is the newest available version[4] which contains fixes for most microarchitectural side-channels against SGX. We use Intel's SGX SDK version 2.8 in all measurements ans always use SGX hardware-mode. For benchmarks that require a network client, we use a dual socket AMD EPYC 7281 machine with 32 cores connected via a switched 1 GBit/s network to the SGX machine.

5.2 Security Discussion

With sgx-dl, enclaves gain a way to load code into themselves after creation. On first look, this might enable attackers to intercept and replace the to-be-loaded code with their own malicious code. To defend against this attack, the sgx-dl library expects a pointer to an ELF file that lies in enclave memory. The enclave first has to load it into enclave memory, after which it can calculate a hashsum and compare it against a known value that is either embedded into the enclave or delivered via a secure channel. If the hashsum comparison fails then the ELF file is not accepted and unloaded from the enclave memory. Instead of a hashsum, the ELF file could

[4] 3.20201110.0ubuntu0.18.04.2

also be signed with a matching public key embedded into the enclave. Encrypting the ELF file is also possible, if confidential code is needed. To add trust into the source of the ELF file, it can be obtained via a secure network connection established between the enclave and a trusted third party (such as a client or another server). The TLS protocol, with the use of certificates on both sides of the connection, provides strong security guarantees and with remote attestation [29], trust in the enclave can be established.

When wanting to call static enclave functions from dynamic functions, sgx-dl needs to inspect the enclave file itself. This is protected by the mechanism presented in §3.2 in which the SThash is backed into the enclave and provided to sgx-dl before loading it. Before calling a function and after patching, sgx-dl makes sure to delete all added functions and objects that are not referenced in any way. This is done to clean out any unused code to reduce the TCB of the enclave. Furthermore, this ensures that superseeded patches are no longer present inside the enclave. An enclave that has been patched multiple times with patches only changing existing functions will not have old copies of those functions and is identical to an newer enclave that only applied the latest patches. Using sgx-dl inside an enclave increases its TCB slightly as the in-enclave part adds 4143 lines of C code[5].

5.3 Microbenchmarks

To measure the overhead of sgx-dl, we evaluate the performance of adding and loading functions into an enclave. Our microbenchmark adds 10,000 functions to an enclave. All added functions are identical except for their name and consist of the C code shown here:

```
void *__funcWXYZ(void *args) { return NULL; }
```

We use `rdtscp` inside the enclave to measure elapsed cycles and convert them to time using the base frequency of our processor. We execute this measurement 1,000 times, which results in an average time of 229.7 µs to add one function to an enclave. Adding functions is not done often so this is an acceptable overhead. After adding the functions, we load all 10,000 functions as our next measurement in the same way. This measurement, also executed 1,000 times, results in an average time of 55.1 µs to load one function into an enclave, which is also an acceptable overhead. Calling a function loaded by sgx-dl incurs an overhead that scales linearly with the number of dependencies it has as sgx-dl has to check that all dependencies are loaded before calling.

5.4 Macrobenchmarks

To evaluate the overhead of sgx-dl with real workloads, we integrate sgx-dl into different SGX applications: The database management system STANlite [48], the data audit and leakage prevention system LibSEAL [8], and the actor framework *E*Actors [47]. All benchmarks were run without pre-loading the dynamic functions: the first invocation of the dynamic function during the benchmark will load the function. Functions are, however, added before the benchmark starts. We also evaluate the hot-patching functionality of sgx-dl by replacing a function during enclave execution.

The applications represent three different ways on how sgx-dl can be used to enhance secure cloud applications. With STANlite, sgx-dl is used to make the main SQL processing engine updatable, which shifts the main processing to dynamic code. In LibSEAL, a module is loaded which contains the main functionality, but the bulk of the request processing is done by the base enclave. Lastly, in *E*Actors, all business logic is dynamically loaded and only actor scheduling is left to the base enclave. While none of our evaluated systems are distributed, there is no restriction on using sgx-dl in a distributed system. With the versioning mechanism in sgx-dl, a cluster can ensure all enclaves are on the same code version.

5.5 STANlite

STANlite [48] is an SGX-aware database management system build on SQLite[6]. It uses a custom *Virtual Memory Engine* to externalize the data stored by the database from the small EPC to the normal main memory of the system. The data is transparently encrypted and hashed to ensure confidentiality and integrity which makes STANlite suited for use on untrusted cloud platforms.

We integrate sgx-dl into STANlite and turn the SQL VM ($\approx$ 43 KiB compiled) of SQLite into dynamic functions. The SQL VM is responsible for executing parsed SQL statements and is the core of the `sqlite3_step` function that evaluates SQL statements[7]. This allows updating the SQL VM to fix bugs without having to restart the enclave. In total, ten functions and six objects are dynamically loaded with a combined size of 44347 and 70 bytes, respectively.

Figure 3 shows the execution time of each test in the SQLite *Speedtest1*[8] test suite. The baseline is the unmodified STANlite running with the `--I` configuration (see §4.B in [48]). For Speedtest1 we used the default payload of SQL records with a relative test size value of 2000, same as in the STANlite paper. All measurements are executed five times and the average is used to calculate the execution times. The measurement error is negligible and not shown. As can be seen, in a macrobenchmark scenario, sgx-dl does not impose significant overhead: the maximum execution time increase is of only 4.7%, in test `140`[9]; the average overhead is 0.5%.

Hot-patching in STANlite. To show the applicability of sgx-dl for code updates, we take our modified STANlite and build a small application that issues `SELECT` queries on a table with 10,000,000 rows. The benchmark works as follows: The application enters the enclave once and then issues queries as fast as possible. After each successful query, a global counter outside the enclave is increased. A thread outside of the enclave is recording the counter value difference periodically every 1,000,000 cycles.

After some time, the enclave is stopped and replaced by a new enclave. For this, we implement a state saving and loading mechanism for STANlite: We save a pointer to the untrusted memory hosting the database data together with some meta-data and a MAC over the memory and the meta-data. The application enters the new enclave, retrieves the pointer and meta-data and verifies the MAC to complete the state transfer. Afterwards, it again issues queries as fast as possible. From a database perspective this can be equated to a cold-restart. For sgx-dl, no state saving and enclave

[5] All numbers were obtained using David A. Wheeler's SLOCcount.

[6] https://www.sqlite.org/

[7] https://www.sqlite.org/c3ref/step.html

[8] STANlite is based on SQLite 3.18.2 which provides that speedtest1 version.

[9] 10 SELECTS, LIKE, unindexed

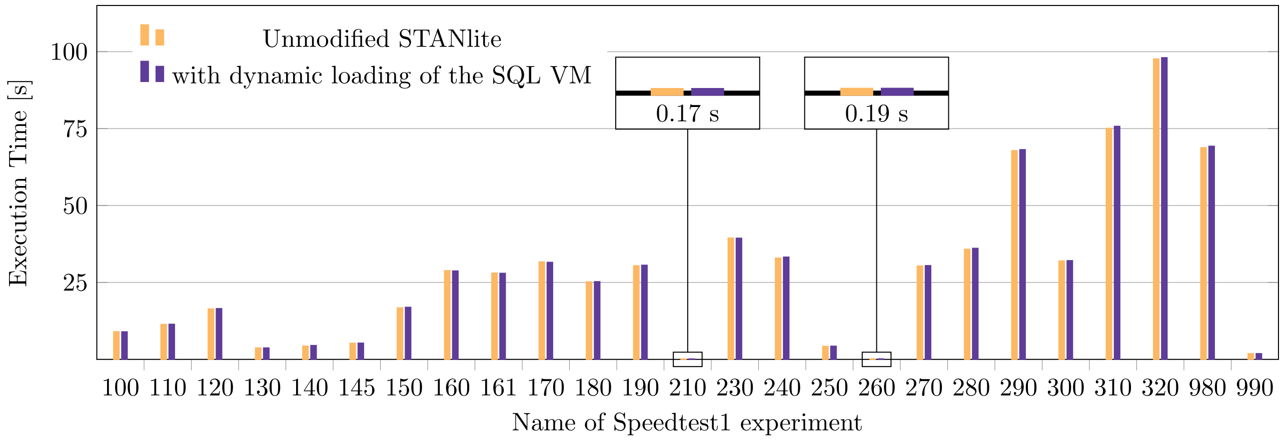


Figure 3: Execution times for the SQLite Speedtest1 test suite for both unmodified STANlite and with dynamically loaded SQL VM. Lower is better, error is negligible.

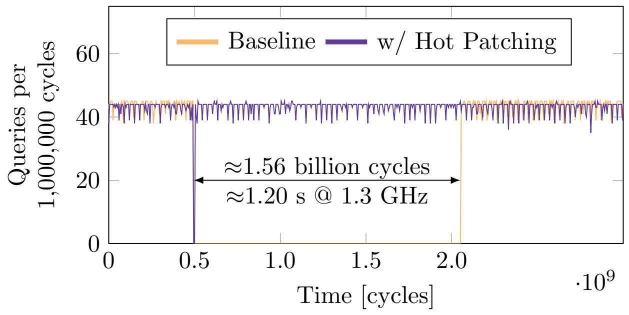


Figure 4: Updating the SQL VM in STANlite by restarting the enclave versus hot-patching with sgx-dl.

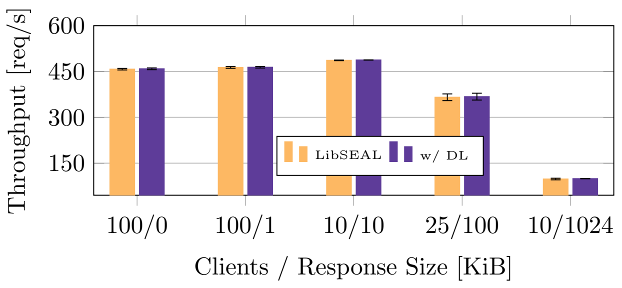


Figure 5: Performance comparison between unmodified LibSEAL and LibSEAL with dynamic loading with Apachebench for various response sizes and client counts.

restart is needed as the SQL-VM is hot-patched between the processing of two queries. The hot-patch replaces the 42 KiB byte big `sqlite3VdbeExec` function with a new version of similar size.

Figure 4 shows the number of executed queries (y-axis) plotted against time (x-axis). We aligned both approaches on the start of the enclave stopping and the start of the hot-patching. As can be seen, the dynamic code loaded by sgx-dl achieves a similar number of queries per sample rate compared to the baseline. Furthermore, in the case of the baseline, the enclave has to be stopped and restarted. The performance thus drops to zero during this process. From last execution in the old enclave to first execution in the new enclave it takes ≈ 1.56 billion cycles, or ≈ 1.20 s on our 1.3 GHz processor, whereas the hot-patched code is ready after ≈ 4 ms. In conclusion, sgx-dl makes hot-patching possible with negligible overhead and downtime compared to the baseline.

5.6 LibSEAL

LibSEAL [8] is a drop-in replacement for TLS libraries (such as OpenSSL/LibreSSL). It implements a non-repudiable audit log inside an enclave to detect violations of the integrity of internet services. For example, LibSEAL can help clients of a cloud storage service to detect and prove the corruption of their online data. LibSEAL allows developers to implement application-specific audit modules and therefore has to provide a tailored enclave for each application.

We integrate sgx-dl into LibSEAL to turn the LibSEAL enclave into a generic enclave that can load arbitrary auditing modules. In our evaluation, we use LibSEAL in combination with the Apache HTTPD web server [60]. The implemented module prevents data disclosure by inspecting the returned traffic from the web server and consists of five functions that are dynamically loaded with a total size of 600 byte. We measure the number of answered requests per second with ApacheBench [59] for responses of different sizes: 0 B, 1 KiB, 10 KiB, 100 KiB and 1024 KiB (excl. HTTP headers).

Figure 5 shows the results of our evaluation for LibSEAL with our audit module statically linked into the enclave versus LibSEAL with the audit module dynamically loaded. We measure various response sizes and include error bars that show the standard deviation over five measurements. We only show the peak performance that was achieved and the respective client count. Performance stagnates while latency increases when increasing the number of clients above this number. As can be seen in the figure, performance of the baseline is comparable to our modified version with a maximum

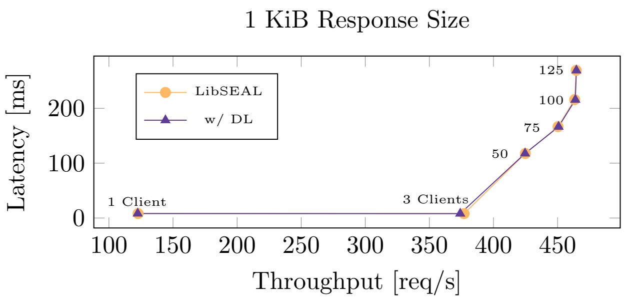


Figure 6: Latency vs throughput for response size 1 KiB and various client numbers. Lower right is better.

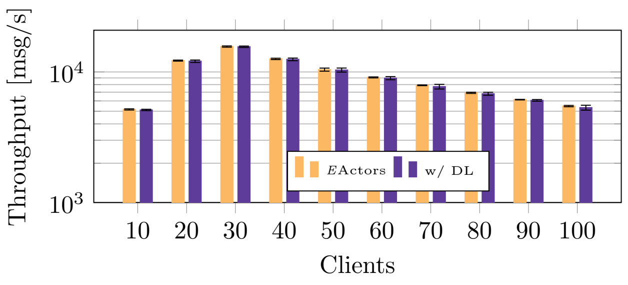


Figure 7: XMPP throughput in messages per second for various client numbers. Error bars show standard deviation over ten runs. Higher is better, y-axis is log scale.

overhead of less than 1% in all cases. The main reason for this behavior is that most of the work during a request is done by the TLS part of LibSEAL and not by the auditing module. As only that module is dynamically loaded, the overhead of using dynamic loading on the module is negligible.

Figure 6 shows the throughput development when increasing the number of clients for a response size of 1 KiB. As can be seen, with 125 clients the throughput peaks at 464 req/s. Latency rises with increasing number of clients. In this scenario, the processor is the bottleneck. With sgx-dl, the throughput develops the same as the baseline when increasing the number of clients.

5.7 *E*Actors

*E*Actors [47] is a framework that enables use of the actor pattern in SGX. Actors are small, stateless execution units that communicate via message passing. With *E*Actors, actors can be distributed over multiple enclaves. *E*Actors targets highly parallel and performant cloud applications such as a message broker or a chat server.

We integrate sgx-dl into *E*Actors to load the actual actor code dynamically instead of it being statically built into the enclave. This allows the generation of a generic enclave that can load the specific actor code on demand. The base enclave only contains the actor scheduler and message passing framework. Furthermore, with sgx-dl, the first step for dynamic reconfiguration of the system is made, i.e., moving actors from one enclave to another, similar to the concept of mobile agents [44]. In total eight functions and twelve objects are dynamically loaded with a combined size of 3938 and 99 bytes, respectively.

For our evaluation, we choose the XMPP benchmark presented in the original *E*Actors paper: An XMPP server implemented as actors and a client application based on *libstrophe* [38]. We use the EA/3 case of the original benchmark, that is three actors: one trusted XMPP actor with its own worker thread as well as two untrusted network reader/writer actors with one worker. We evaluate different numbers of clients exchanging messages through the server. Each test is run ten times for one minute each.

Figure 7 shows the results of the base system compared to a system in which the XMPP actor loads its code dynamically. As can be seen, with increasing client numbers, the throughput first goes up and peaks for 30 clients and then goes down due to increased load on the system. However, the overhead of using sgx-dl to dynamically load the actor code is again small, ranging from 2.7% (100 clients) to 0.4% (30 clients) with an average of 1.3%.

6 CONCLUSION

In this paper, we presented sgx-dl, a novel system for dynamically loading and hot-patching code in Intel SGX enclaves with versioning support. sgx-dl is the first system to utilize SGXv2 features for adding code safely to an enclave. Furthermore, sgx-dl is the first system to enable hot-patching of code in an enclave for restartless updates. Dynamically loaded and hot-patched code can be attested through sgx-dl's versioning mechanism. Our evaluation shows that sgx-dl adds an overhead of less than 5% on average in our use-cases. Furthermore, hot-patching a stateful database system with sgx-dl reduced the outage time to 4 ms compared to 1.2 s when restarting it. The source code of sgx-dl is publicly available at https://github.com/ibr-ds/sgx-dl.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their helpful comments. We would like to thank the authors of STANLite [48], LibSEAL [8] and *E*Actors [47] for providing their source code and to allow integration of sgx-dl for evaluation purposes. This work was supported by the German Research Foundation (DFG) under grant no. KA 3171/9-1.

REFERENCES

[1] 2019. Azure Confidential Computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Accessed on 2021-03-21.
[2] 2019. Azure Confidential Computing. https://docs.microsoft.com/en-us/azure/automanage/automanage-hotpatch. Accessed on 2021-03-21.
[3] 2019. Hot Patching SQL Server Engine in Azure SQL Database. https://techcommunity.microsoft.com/t5/azure-sql/hot-patching-sql-server-engine-in-azure-sql-database/ba-p/849700. Accessed on 2021-03-21.
[4] 2021. Canonical Livepatch Service. https://ubuntu.com/security/livepatch. Accessed on 2021-03-21.
[5] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *Network and Distributed System Security Symposium (NDSS'19)*.
[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[7] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EUROSYS '09)*.

[8] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2018. LibSEAL: Revealing Service Integrity Violations Using Trusted Execution. In *Proceedings of the Thirteenth EuroSys Conference (EUROSYS'18)*.

[9] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*.

[11] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*.

[12] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. 2019. DR. SGX: automated and adjustable side-channel protection for SGX using data location randomization. In *35th Annual Computer Security Applications Conference (AC-SEC'19)*.

[13] Sergey Bratus, James Oakley, Ashwin Ramaswamy, Sean W. Smith, and Michael E. Locasto. 2010. Katana: Towards Patching As a Runtime Part of the Compiler-Linker-Loader Toolchain. *International Journal of Secure Software Engineering (IJSSE)* (2010).

[14] Stefan Brenner, Colin Wulf, Matthias Lorenz, Nico Weichbrodt, David Goltzsche, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 15th International Middleware Conference (MIDDLEWARE)*.

[15] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 69–76.

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P'19)*.

[17] Alibaba Cloud. 2021. ECS Bare Metal Instance. https://www.alibabacloud.com/product/ebm. Accessed on 2021-04-05.

[18] Jonathan Corbet. 2014. The initial kGraft submission. https://lwn.net/Articles/596854/. Accessed on 2019-10-31.

[19] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[20] Jon Fingas. 2014. Dropbox Bug Wipes Some Users' Files From the Cloud. https://www.engadget.com/2014/10/13/dropbox-selective-sync-bug/. Accessed on 2019-11-02.

[21] Fortanix. 2019. Fortanix Enclave Development Platform. https://edp.fortanix.com/. Accessed on 2019-09-13.

[22] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. 2012. Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*.

[23] Sean Hollister. 2011. Gmail accidentally resetting accounts, years of correspondence vanish into the cloud? https://www.engadget.com/2011/02/27/gmail-accidentally-resetting-accounts-years-of-correspondence-v/. Accessed on 2019-11-02.

[24] Intel. 2017. Intel Software Guard Extensions (SGX) Protected Code Loader (PCL). https://github.com/intel/linux-sgx-pcl. Accessed on 2019-08-07.

[25] Intel. 2018. Intel Developer Zone - L1 Terminal Fault Software Guidance. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault. Accessed on 2019-10-31.

[26] Intel. 2018. Intel Software Guard Extensions SDK for Linux. https://01.org/intel-softwareguard-extensions. Accessed on 2019-08-07.

[27] Intel. 2018. Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass (CVE-2017-5753) Side Channel Exploits. https://software.intel.com/sites/default/files/180204_SGX_SDK_Developer_Guidance_v1.0.pdf. Accessed on 2019-10-30.

[28] Pratheek Karnati. 2018. Data-in-use protection on IBM Cloud using Intel SGX . https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx. Accessed on 2021-04-05.

[29] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating Remote Attestation with Transport Layer Security. (2018). http://arxiv.org/abs/1801.05863

[30] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203* (2018).

[31] Sebastian Krieter, Tobias Thiem, and Thomas Leich. 2019. Using Dynamic Software Product Lines to Implement Adaptive SGX-enabled Systems. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'19)*.

[32] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2020. With Great Power Comes Great Leakage: Software-based Power Side-Channel Attacks on x86. (2020).

[33] LSDS Team, Imperial College London. 2018. github: sgx-lkl. https://github.com/lsds/sgx-lkl. Accessed on 2019-10-30.

[34] Sébastien Martinez, Fabien Dagnat, and Jérémy Buisson. 2013. Prototyping DSU Techniques Using Python. In *5th Workshop on Hot Topics in Software Upgrades (HotSWUp'13)*.

[35] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security'17)*.

[36] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT'14)*.

[37] Microsoft. 2019. Hot patching SQL Server Engine in Azure SQL Database. https://azure.microsoft.com/en-us/blog/hot-patching-sql-server-engine-in-azure-sql-database/. Accessed on 2021-03-21.

[38] Jack Moffitt. 2018. libstrophe - An XMPP library for C. http://strophe.im/libstrophe/. Accessed on 2019-10-31.

[39] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*.

[40] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. 2018. github: spectre-attack-sgx. https://github.com/lsds/spectre-attack-sgx. Accessed on 2019-10-30.

[41] Oracle. 2021. Oracle RDBMS Online Patching Aka Hot Patching. https://support.oracle.com/knowledge/Oracle%20Database%20Products/761111_1.html. Accessed on 2021-03-21.

[42] Meni Orenbach, Andrew Baumann, and Mark Silberstein. 2020. Autarky: closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[43] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. 2019. CoSMIX: a compiler-based system for secure memory instrumentation and execution in enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*.

[44] Vu Anh Pham and Ahmed Karmouch. 1998. Mobile Software Agents: An Overview. *IEEE Communications Magazine* (1998).

[45] Josh Poimboeuf. 2014. kpatch: dynamic kernel patching. https://lwn.net/Articles/597123/.

[46] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB – A Secure Database using SGX. In *To appear in the Proceedings of the IEEE Symposium on Security & Privacy (2018)*. https://www.microsoft.com/en-us/research/publication/enclavedb-a-secure-database-using-sgx/

[47] Vasily A Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. 2018. EActors: Fast and flexible trusted computing using SGX. In *Proceedings of the 19th International Middleware Conference (Middleware'18)*.

[48] Vasily A Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rüdiger Kapitza. 2018. STANlite–a database engine for secure data processing at rack-scale level. In *Proceedings of the Sixth International Conference on Cloud Engineering (IC2E'18)*.

[49] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*.

[50] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. *arXiv:1905.05726* (2019).

[51] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Network and Distributed System Security Symposium (NDSS'17)*.

[52] Simon Sharwood. 2017. GitLab.com Melts Down After Wrong Directory Deleted, Backups Fail. https://www.theregister.co.uk/2017/02/01/gitlab_data_loss/?mt=1486066707837. Accessed on 2019-11-02.

[53] Mingwei Shih. 2019. *Securing Intel SGX against Side-channel Attacks via Load-time Synthesis.* Ph.D. Dissertation. Georgia Institute of Technology.

[54] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium (NDSS'17)*.

[55] Rodolfo Silva, Pedro Barbosa, and Andrey Brito. 2017. DynSGX: A Privacy Preserving Toolset for Dynamically Loading Functions into Intel (R) SGX Enclaves. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom'17)*.

[56] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1–15.

[57] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security'16)*.

[58] The Rust Embedded Resources Team. 2019. A no_std Rust Environment. https://rust-embedded.github.io/book/intro/no-std.html. Accessed on 2019-09-13.

[59] The Apache Software Foundation. 2019. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/current/programs/ab.html. Accessed on 2019-10-31.

[60] The Apache Software Foundation. 2019. Apache HTTP Server Project. https://httpd.apache.org/. Accessed on 2019-08-12.

[61] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC'17)*.

[62] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium. (USENIX Security'18)*.

[63] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[64] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security'17)*.

[65] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. https://cacheoutattack.com/.

[66] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*.

[67] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security (ESORICS'16)*.

[68] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (IEEE S&P'15)*.

6 Conclusions

At the time of writing, Intel has dropped support for SGX on desktop platforms and 11th (Rocket Lake) generation processors and newer are not capable of launching SGX enclaves any more. At first glance, this seems to invalidate the work presented in this thesis as SGX is used in all the work presented here but in reality, SGX was never a good fit for the client side. When Intel released SGX, it provided a few sample application ideas on how SGX can be used to secure client side applications. However, academia and the adoption by companies [28, 79] have shown that SGX should have never been a client side technology to begin with and with the release of the Ice Lake-SP platform Intel has finally brought SGX to its datacenter line of processors.

The work shown in this thesis all focuses on applications running on the server side, e.g., the cloud, on which SGX is now available. With sgx-perf a performance profiling framework has been presented that lends itself perfectly to analyse server-side applications and helps developers optimize their software on the next generation of processors. While the low EPC limit is a thing of the past, ECall and OCall performance is still a metric important for optimization, especially when new μ-code versions that change the performance will be released.

One way to increase application performance, is to parallelize workloads, e.g. by utilizing multiple threads. AsyncShock has shown that multithreading is now a potential threat vector as it can be used to break or exfiltrate data from SGX applications. Being one of the first attacks on applications running on top of SGX, it has shown that mitigating not only microarchitectural side-channels but also application side-channels is important. Furthermore, even though the implementation of the EPC has changed, AsyncShock still works as the underlying mechanism of how page faults are handled has not changed.

One way to harden against novel threats in the cloud is to ship frequent application updates. With sgx-dl a novel framework for dynamic loading and hot-patching has been released which is optimal for use in the cloud. sgx-dl enables dynamic applications in the cloud with low latency and with hot-patching sgx-dl helps to keep the application downtime minimal.

6.1 Outlook

With only having been available since 2016, Intel SGX is still a young technology. Intel has already extended the capabilities of SGX quite a bit in the past, but it has shown no signs of stopping yet in the confidential computing area. The latest SGX extension *AEX-Notify* [29, 41] shows a promising way for enclaves to defend against attackers trying to single-step enclave execution. With Total Memory Encryption (TME) and Total Memory Encryption

- Multi-Key (TME-MK) [46], Intel has already presented its version of AMD's SME and as soon as it's released, a new target for microarchitectural side-channels will exist. Furthermore, Intel has also presented its version of AMD's SEV, namely Intel Trust Domain Extensions (TDX) [47] but nothing with regard to its availability is known yet. AMD has enhanced SEV with SNP [3] but still focuses on securing whole virtual machines instead of partitioning applications. ARM is also working on Confidential Compute Architecture (CCA) [8] which combines TrustZone and virtualization with memory encryption in its Armv9-A architecture. Judging by these announcements of upcoming features, trusted execution should be viewed as an important aspect of general purpose computing as Intel, AMD and ARM all invest in the enhancement of their respective technologies.

Bibliography

[1] Abbas Acar et al. “A Survey on Homomorphic Encryption Schemes: Theory and Implementation”. In: *ACM Computing Surveys (Csur)* 51.4 (2018). DOI: 10.1145/3214303.

[2] *AMD Secure Encrypted Virtualization (SEV)*. https://developer.amd.com/sev/. Accessed on 2024-02-15. 2022.

[3] *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Accessed on 2024-02-15. 2020.

[4] *An update on 3rd Party Attestation*. https://www.intel.com/content/www/us/en/developer/articles/technical/an-update-on-3rd-party-attestation.html. Accessed on 2024-02-15. 2018.

[5] Ittai Anati et al. “Innovative technology for CPU based attestation and sealing”. In: *2nd international workshop on hardware and architectural support for security and privacy (HASP ’13)*. 2013.

[6] *Apache Teaclave (incubating)*. https://teaclave.apache.org/. Accessed on 2024-02-15. 2023.

[7] ARM. *Layered Security for Your Next SoC*. https://www.arm.com/products/silicon-ip-security. Accessed on 2024-02-15. 2022.

[8] *Arm Confidential Compute Architecture*. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed on 2024-02-15. 2023.

[9] Sergei Arnautov et al. “SCONE: Secure Linux Containers with Intel SGX”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16)*. 2016.

[10] Jeff Arnold and M. Frans Kaashoek. “Ksplice: Automatic Rebootless Kernel Updates”. In: *Proceedings of the 4th ACM European Conference on Computer Systems (EUROSYS ’09)*. 2009. DOI: 10.1145/1519065.1519085.

[11] *Asylo*. https://asylo.dev/. Accessed on 2024-02-15. 2023.

[12] Multiple Authors. *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on 2024-02-15. 2023.

[13] Multiple Authors. *Profiler - Home Assistant*. https://www.home-assistant.io/integrations/profiler/. Accessed on 2024-02-15. 2023.

[14] Multiple Authors. *The Python Profilers*. https://docs.python.org/3/library/profile.html. Accessed on 2024-02-15. 2023.

[15] Michael Backes et al. "You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code". In: *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. 2014.

[16] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14*. 2014, pp. 267–283.

[17] Andrea Biondo et al. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium (USENIX Security '18)*. 2018.

[18] Nikita Borisov et al. "Fixing Races for Fun and Profit: How to Abuse Atime". In: *14th Conference on USENIX Security Symposium - Volume 14 (SSYM '05)*. 2005, p. 20.

[19] Ferdinand Brasser et al. "DR. SGX: Automated and adjustable side-channel protection for SGX using data location randomization". In: *35th Annual Computer Security Applications Conference (ACSAC '19)*. 2019. doi: `10.1145/3359789.3359809`.

[20] Ferdinand Brasser et al. "DR. SGX: automated and adjustable side-channel protection for SGX using data location randomization". In: *35th Annual Computer Security Applications Conference (AC-SEC'19)*. 2019.

[21] Sergey Bratus et al. "Katana: Towards Patching As a Runtime Part of the Compiler-Linker-Loader Toolchain". In: *International Journal of Secure Software Engineering (IJSSE '10)* (2010). doi: `10.4018/jsse.2010070101`.

[22] Stefan Brenner et al. "Securekeeper: confidential zookeeper using intel sgx". In: *17th International Middleware Conference (Middleware '16)*. 2016.

[23] Ernie Brickell and Jiangtao Li. "Enhanced privacy ID from bilinear pairing for hardware authentication and attestation". In: *International Journal of Information Privacy, Security and Integrity* 2 (2011).

[24] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation". In: *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19*. 2019. doi: `10.1145/3319535.3354216`.

[25] *Canonical Livepatch Service*. `https://ubuntu.com/security/livepatch`. Accessed on 2024-02-15. 2021.

[26] Guoxing Chen et al. "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races". In: *2018 IEEE Symposium on Security and Privacy (S&P '18)*. 2018. doi: `10.1109/SP.2018.00024`.

[27] Guoxing Chen et al. "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P '19)*. 2019.

[28] *Confidential Computing*. `https://www.fortanix.com/products/confidential-computing`. Accessed on 2024-02-15. 2022.

[29] Scott Constable et al. “AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves”. In: *32nd USENIX Security Symposium*. 2023, pp. 4051–4068.

[30] Jonathan Corbet. *The initial kGraft submission*. `https://lwn.net/Articles/596854/`. Accessed on 2024-02-15. 2014.

[31] Drew Dean and Alan J. Hu. “Fixing Races for Fun and Profit: How to Use Access(2)”. In: *13th Conference on USENIX Security Symposium - Volume 13 (SSYM ’04)*. 2004, p. 14.

[32] DJ Delorie. *How the GNU C Library handles backward compatibility*. `https://developers.redhat.com/blog/2019/08/01/how-the-gnu-c-library-handles-backward-compatibility`. Accessed on 2024-02-15. 2019.

[33] Anders T Gjerdrum et al. “Performance of Trusted Computing in Cloud Infrastructures with Intel SGX”. In: *7th International Conference on Cloud Computing and Services Science (CLOSER)*. 2017.

[34] ej-technoloies GmbH. *Java Profiler - JProfiler*. `https://www.ej-technologies.com/products/jprofiler/overview.html`. Accessed on 2024-02-15. 2023.

[35] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009.

[36] Shay Gueron. “Memory Encryption for General-Purpose Processors”. In: *IEEE Security & Privacy* (2016). doi: `10.1109/MSP.2016.124`.

[37] Matthew Hoekstra et al. “Innovative Instructions to Create Trustworthy Software Solutions”. In: *2nd international workshop on hardware and architectural support for security and privacy (HASP ’13)*. 2013.

[38] Shohreh Hosseinzadeh et al. “Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization”. In: *3rd Workshop on System Software for Trusted Execution (SysTEX ’18)*. 2018. doi: `10.1145/3268935.3268940`.

[39] *Hotpatch for virtual machines*. `https://learn.microsoft.com/en-us/windows-server/get-started/hotpatch`. Accessed on 2024-02-15. 2023.

[40] Intel. https://www.intel.com/content/www/us/en/content-details/782158/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html. Accessed on 2024-02-15. 2023.

[41] Intel. *Asynchronous Enclave Exit Notify and the EDECCSSA User Leaf Function*. `https://www.intel.com/content/www/us/en/content-details/736463/white-paper-asynchronous-enclave-exit-notify-and-the-edeccssa-user-leaf-function.html`. Accessed on 2024-02-15. 2022.

[42] Intel. *Intel SGX and Side-Channels*. `https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-enhanced-data-protection.html`. Accessed on 2024-02-15. 2027.

[43] Intel. *Intel Software Guard Extensions Programming Reference, Revision 2*. `https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf`. Accessed on 2024-02-15. 2014.

[44] Intel. *Intel Software Guard Extensions (SGX) Protected Code Loader (PCL)*. `https://github.com/intel/linux-sgx-pcl`. Accessed on 2024-02-157. 2017.

[45] Intel. *Intel Software Guard Extensions Developer Reference for Linux OS*. `https://download.01.org/intel-sgx/sgx-linux/2.17/docs/Intel_SGX_Developer_Reference_Linux_2.17_Open_Source.pdf`. Accessed on 2024-02-15. 2022.

[46] *Intel Total Memory Encryption*. `https://www.intel.de/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf`. Accessed on 2024-02-15. 2021.

[47] *Intel Trust Domain Extensions*. `https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html`. Accessed on 2024-02-15. 2023.

[48] *Intel Trusted Execution Technology*. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf`. Accessed on 2024-02-15. 2012.

[49] Intel. *Intel VTune Profiler*. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`. Accessed on 2024-02-15. 2023.

[50] *Intel(R) Software Guard Extensions for Linux* OS*. `https://github.com/intel/linux-sgx`. Accessed on 2024-02-15. 2022.

[51] *Information technology – Trusted platform module library – Part 1: Architecture*. Standard. Geneva, CH: ISO/IEC JTC 1 Information technology, Aug. 2015.

[52] Yeongjin Jang et al. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack". In: *2nd Workshop on System Software for Trusted Execution (SysTEX '17)*. 2017.

[53] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *arXiv:1801.01203* (2018).

[54] Robert Krahn et al. "TEEMon: A continuous performance monitoring framework for TEEs". In: *21st International Middleware Conference (Middleware '20)*. 2020.

[55] Sebastian Krieter, Tobias Thiem, and Thomas Leich. "Using Dynamic Software Product Lines to Implement Adaptive SGX-enabled Systems". In: *13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '19)*. 2019.

[56] Fan Lang et al. "MoLE: Mitigation of Side-channel Attacks against SGX via Dynamic Data Location Escape". In: *38th Annual Computer Security Applications Conference (ACSAC '22)*. 2022. doi: `10.1145/3564625.3568002`.

[57] Shan Lu et al. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”. In: *P13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’08)*. 2008. doi: 10.1145/1346281.1346323.

[58] Mohammad Mahhouk, Nico Weichbrodt, and Rüdiger Kapitza. “SGXoMeter: Open and Modular Benchmarking for Intel SGX”. In: *14th European Workshop on Systems Security (EuroSec ’21)*. 2021.

[59] Giovanni Mazzeo et al. “SGXTuner: Performance Enhancement of Intel SGX Applications via Stochastic Optimization”. In: *IEEE Transactions on Dependable and Secure Computing* (2021).

[60] Frank McKeen et al. “Innovative Instructions and Software Model for Isolated Execution”. In: *2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP ’13)*. 2013.

[61] Microsoft. *Hot patching SQL Server Engine in Azure SQL Database*. https://azure.microsoft.com/en-us/blog/hot-patching-sql-server-engine-in-azure-sql-database/. Accessed on 2024-02-15. 2019.

[62] Susan Moore. *Gartner Says More Than Half of Enterprise IT Spending in Key Market Segments Will Shift to the Cloud by 2025*. https://www.gartner.com/en/newsroom/press-releases/2022-02-09-gartner-says-more-than-half-of-enterprise-it-spending. Accessed on 2024-02-15. 2022.

[63] Kit Murdock et al. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *41st IEEE Symposium on Security and Privacy (S&P ’20)*. 2020.

[64] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. “Can Homomorphic Encryption Be Practical?” In: *3rd ACM Workshop on Cloud Computing Security Workshop (CCSW ’11)*. 2011. doi: 10.1145/2046660.2046682.

[65] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *28th ACM SIGPLAN Conference on Programming Language Design and Implementation (SIGPLAN ’07)*. 2007, pp. 89–100. doi: 10.1145/1250734.1250746.

[66] Hyunyoung Oh et al. “TRUSTORE: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA”. In: *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS ’20)*. 2020. doi: 10.1145/3372297.3417265.

[67] Oleksii Oleksenko et al. “Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks”. In: *2018 Usenix Annual Technical Conference (USENIX ATC ’18)*. 2018.

[68] Oracle. *JDK Mission Critical*. https://www.oracle.com/java/technologies/jdk-mission-control.html. Accessed on 2024-02-15. 2023.

[69] Oracle. *Oracle RDBMS Online Patching Aka Hot Patching*. https://support.oracle.com/knowledge/Oracle%20Database%20Products/761111_1.html. Accessed on 2024-02-15. 2021.

[70] Meni Orenbach et al. “Eleos: ExitLess OS Services for SGX Enclaves”. In: *12th European Conference on Computer Systems (EuroSys ’17)*. 2017.

[71] Mathias Payer and Thomas R. Gross. “Hot-patching a web server: A case study of ASAP code repair”. In: *2013 Eleventh Annual Conference on Privacy, Security and Trust*. 2013.

[72] Josh Poimboeuf. *kpatch: dynamic kernel patching*. `https://lwn.net/Articles/597123/`. Accessed on 2024-02-15. 2014.

[73] Meghan Rimol. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $500 Billion in 2022*. `https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022`. Accessed on 2024-02-15. 2022.

[74] *Samsung Trusted Boot and TrustZone Integrity Management explained*. `https://www.samsungknox.com/en/blog/samsung-trusted-boot-and-trustzone-integrity-management-explained`. Accessed on 2024-02-15. 2019.

[75] *Scaling Towards Confidential Computing*. `https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf`. Accessed on 2024-02-15. 2019.

[76] Stephan van Schaik et al. *CacheOut: Leaking Data on Intel CPUs via Cache Evictions*. `https://cacheoutattack.com/`. 2020.

[77] Felix Schuster et al. “VC3: Trustworthy Data Analytics in the Cloud using SGX”. In: *2015 IEEE Symposium on Security and Privacy (S&P’15)*. 2015.

[78] Michael Schwarz et al. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *arXiv:1905.05726* (2019).

[79] *SCONE - A Secure Container Environment*. `https://scontain.com/`. Accessed on 2024-02-15. 2022.

[80] Jaebaek Seo et al. “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs”. In: *Network and Distributed System Security Symposium (NDSS’17)*. 2017. doi: `10.14722/ndss.2017.23037`.

[81] Konstantin Serebryany and Timur Iskhodzhanov. “ThreadSanitizer: data race detection in practice”. In: *Workshop on Binary Instrumentation and Applications*. 2009. doi: `10.1145/1791194.1791203`.

[82] LSDS Team, Imperial College London. *github: sgx-lkl*. `https://github.com/lsds/sgx-lkl`. Accessed on 2024-02-15. 2018.

[83] Mingwei Shih. “Securing Intel SGX against Side-channel Attacks via Load-time Synthesis”. PhD thesis. Georgia Institute of Technology, 2019.

[84] Rodolfo Silva, Pedro Barbosa, and Andrey Brito. “DynSGX: A Privacy Preserving Toolset for Dynamically Loading Functions into Intel (R) SGX Enclaves”. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom ’17)*. 2017.

[85] Rohit Sinha et al. “Moat: Verifying Confidentiality of Enclave Programs”. In: *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS ’15)*. 2015, pp. 1169–1184. DOI: `10.1145/2810103.2813608`.

[86] Linus Torvalds. *Re: Very slow clang kernel config* .. `https://lore.kernel.org/lkml/CAHk-=whs8QZf3YnifdLv57+FhBi5_WeNTG1B-suOES=RcUSmQg@mail.gmail.com/`. Accessed on 2024-12-15. 2021.

[87] Linus Torvalds. *Why your Go programs can surprisingly be dynamically linked.* `https://utcc.utoronto.ca/~cks/space/blog/programming/GoWhyNotStaticLinked`. Accessed on 2024-03-15. 2021.

[88] Dan Tsafrir et al. “Portably Solving File TOCTTOU Races with Hardness Amplification.” In: *6th USENIX Conference on File and Storage Technologies (FAST’08)*. 2008, pp. 1–18.

[89] 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.

[90] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *2nd Workshop on System Software for Trusted Execution (SysTEX ’17)*. 2017.

[91] 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.

[92] Jo Van Bulck et al. “LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”. In: *41th IEEE Symposium on Security and Privacy (S&P’20)*. 2020.

[93] 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.

[94] Marcus Völp et al. “Avoiding Leakage and Synchronization Attacks through Enclave-Side Preemption Control”. In: *1st Workshop on System Software for Trusted Execution (SysTEX ’16)*. 2016. DOI: `10.1145/3007788.3007794`.

[95] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. “sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves”. In: *19th International Middleware Conference (Middleware ’18)*. 2018. DOI: `10.1145/3274808.3274824`.

[96] Nico Weichbrodt et al. “AsyncShock: Exploiting Aynchronisation Bugs in Intel SGX Enclaves”. In: *European Symposium on Research in Computer Security (ESORICS ’16)*. 2016.

[97] Nico Weichbrodt et al. “Experience Paper: sgx-dl: Dynamic Loading and Hot-Patching for Secure Applications”. In: *22nd International Middleware Conference (Middleware ’21)*. 2021.

[98] Ofir Weisse, Valeria Bertacco, and Todd Austin. “Regaining Lost Cycles with Hot-Calls: A Fast Interface for SGX Secure Enclaves”. In: *44th Annual International Symposium on Computer Architecture (ISCA ’17)*. 2017.

[99] Weiwei Xiong et al. “Ad Hoc Synchronization Considered Harmful”. In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’10)*. 2010.

[100] 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 (S&P ’15)*. 2015.

[101] Junfeng Yang et al. “Concurrency Attacks”. In: *4th USENIX Workshop on Hot Topics in Parallelism (HotPar ’12)*. 2012.

[102] C. Zhao et al. “On the Performance of Intel SGX”. In: *3th Web Information Systems and Applications Conference (WISA ’16)*. 2016.