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

Middleware '18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-5702-9/18/12…$15.00
https://doi.org/10.1145/3274808.3274824

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 entry-point 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 user-space 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 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.

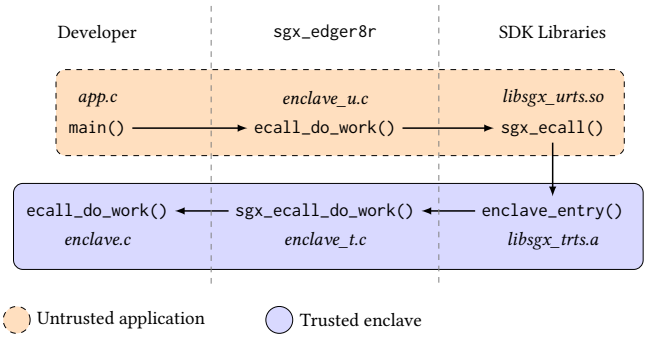

Figure 1: Architecture of an ecall. The developer provides the application and ecall implementations whereas the SDK generates code which uses the URTS and TRTS libraries.

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,

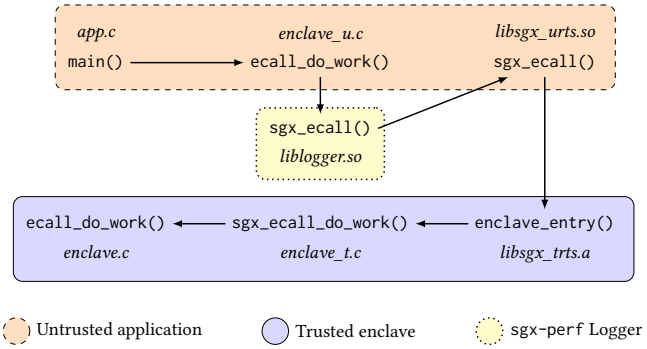

Figure 2: *sgx-perf* tracks ecalls by shadowing the call to `sgx_ecall` so it is called instead of the URTS.

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

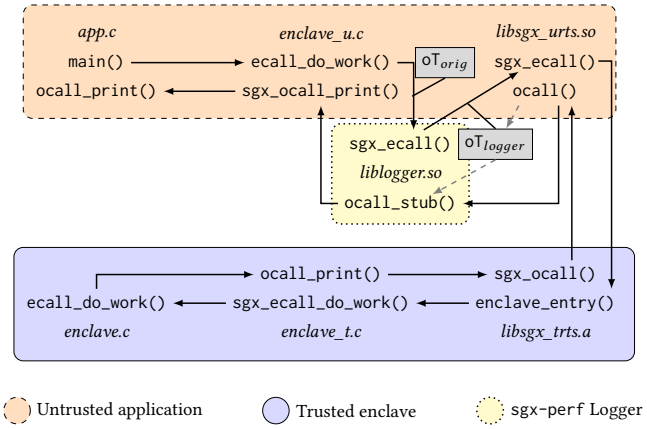

Figure 3: *sgx-perf* rewrites the ocall table $\mathtt{oT}_{orig}$ to its own table $\mathtt{oT}_{logger}$ during ecalls to track ocalls.

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.

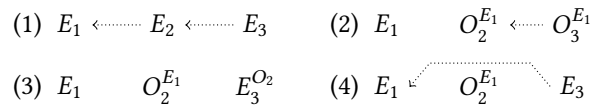

Figure 4: Some example for calls (C) with their direct (C^P) and indirect ($P \leftarrow C$) parents (P).

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.

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

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.

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

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.

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

As before, $\alpha, \beta, \gamma, \delta, \epsilon$ 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 ($\alpha, \beta, \gamma, \delta$, 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.

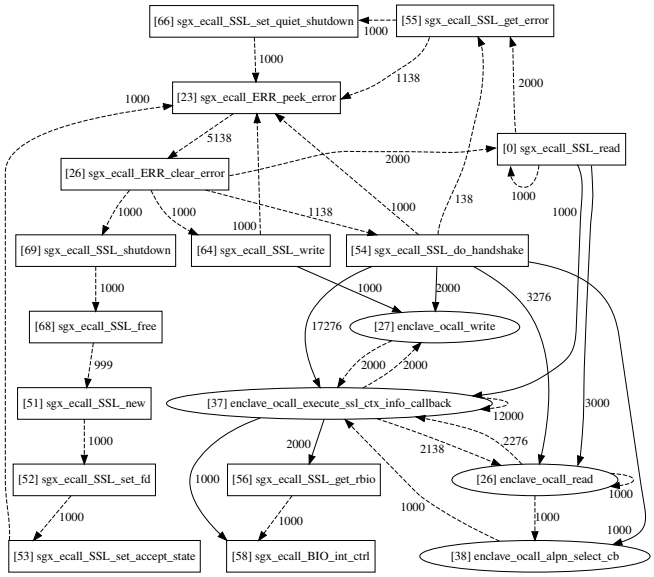

Figure 5: nginx + TaLoS main enclave calls. Square nodes are ecalls, round nodes are ocalls. Solid arrows indicate direct parents, dashed arrows indirect parents. Numbers on edges indicate call count, numbers in brackets indicate call id.

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.

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

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.

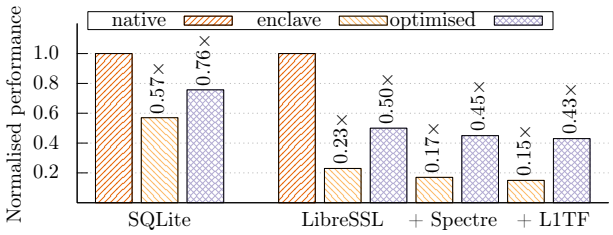

Figure 6: Normalised performance for SQLite and LibreSSL for native, enclavised and optimised execution.

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`:

```
1  void bn_mul_recursive(...) {
2     // ...
3     switch (c1 * 3 + c2) {
4     case -4:
5        ecall_bn_sub_part_words(t, a+n, a, tna, tna-n);
6        ecall_bn_sub_part_words(t+n, b, b+n, tnb, n-tnb);
7        break;
8     // ... Repeated three more times
9     }
10    // ...
11 }
```

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.

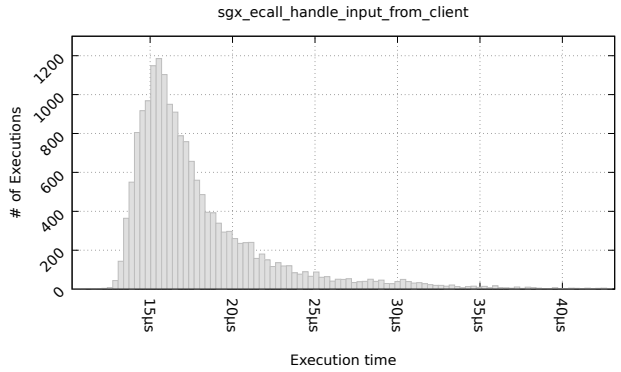

Figure 7: Generated histogram of call execution times for one of SecureKeeper's ecalls grouped into 100 bins.

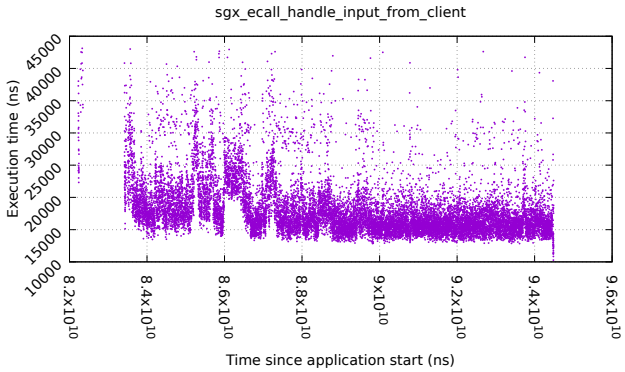

Figure 8: Generated scatter plot of call execution times for one of SecureKeeper's ecalls.

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)*.