SGXoMeter: Open and Modular Benchmarking for Intel SGX

Mohammad Mahhouk
TU Braunschweig, Germany

Nico Weichbrodt
TU Braunschweig, Germany

Rüdiger Kapitza
TU Braunschweig, Germany

ABSTRACT

Intel's Software Guard Extensions (SGX) are currently the most wide-spread commodity trusted execution environment, which provides integrity and confidentiality of sensitive code and data. Thereby, it offers protection even against privileged attackers and various forms of physical attacks. As a technology that only became available in late 2015, it has received massive interest and undergone a rapid evolution. Despite first ad-hoc attempts, there is so far no standardised approach to benchmark the SGX hardware, its associated environment, and techniques that were designed to harden SGX-based applications.

In this paper, we present SGXoMeter, an open and modular framework designed to benchmark different SGX-aware CPUs, μcode revisions, SDK versions and extensions to mitigate side-channel attacks. SGXoMeter provides a set of practical SGX test case scenarios and eases the development of custom benchmarks. Furthermore, we compare it to *sgx-nbench*, the only other SGX application benchmark tool we are aware of, and evaluate their differences. Through our benchmark results, we identified a performance overhead of up to ≈10 times induced between two different SGX-SDK versions for certain workload scenarios.

CCS CONCEPTS

• **Security and privacy → Hardware-based security protocols**; *Trusted computing*; • **General and reference** → Performance.

KEYWORDS

Benchmarking, SGX, Trusted Execution

ACM Reference Format:
Mohammad Mahhouk, Nico Weichbrodt, and Rüdiger Kapitza. 2021. SGXoMeter: Open and Modular Benchmarking for Intel SGX. In *14th European Workshop on Systems Security (EuroSec'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3447852.3458722

1 INTRODUCTION

The development of hardware-based trusted execution environments (TEEs) has gained a lot of interest lately because they offer security and protection from a variety of threats including potentially compromised operating systems, kernel exploits and untrustworthy cloud services [20]. Hardware-based TEEs have found usage in wide range of devices, e.g., ARM's TrustZone [29] is deployed in mobile devices like phones and tablets. Also, personal computers, laptops and servers can be secured using AMD Secure Encrypted Virtualisation [22, 23], and Intel SGX [24, 27].

Intel SGX promises with its isolated memory regions, so called *enclaves*, both confidentiality and integrity protection of the sensitive data and code running inside them against malicious and privileged software. It also provides local and remote attestation mechanisms [21] to ensure the authenticity and integrity of the running enclaves, adding protection against forging attempts. Thus, the utilisation of SGX in cloud services can considerably reduce the customers' reluctance of using them. Furthermore, Intel has released a Software Development Kit (SDK) [18] to ease programming with SGX. It introduces wrappers for low-level instructions and provides a high-level interface that offers multiple functionalities, such as enclave initialisation, and parameterised transition functions into and outside an enclave. The strong security guarantees and the ease of development with the SGX-SDK paved the way to deploy SGX in the mainstream industry, like the confidential clouds of Microsoft Azure [8]. In addition, it enabled the creation of diverse secure applications, such as Signal [10], SecureKeeper [12] and the confidential computing consortium projects [9].

Currently, SGX technology is maturing rapidly through more instruction set changes, continuous releases of new drivers and μcode patches [18, 19]. It has also been extensively analysed how to enhance the security of SGX against malicious activities such as controlled and side-channel attacks [15, 16, 30, 34]. Therefore, Intel is actively updating the SDK releases to address and mitigate these attacks. All of the aforementioned factors have various impact on the overall performance of SGX-secured applications and will remain a challenge for future developments. For example, SDK mitigation patches against Spectre [25] and Foreshadow [13] add extra performance overhead for enclave transitions of up to 2.24× compared to the original costs [32]. Multiple research projects have observed 5.5× performance overhead upon exceeding the L3 cache size and 1000× when the Enclave Page Cache (EPC) limit is exceeded due to the costly enclave paging [11, 12]. However, at the moment there is no easy to use and suitable benchmark tool designed for SGX. A standard application benchmark framework is necessary for both proprietary and research projects. It can inspect the performance overhead in different environment configurations and allows researchers to reproduce their results and compare them to other research projects.

The contributions of this paper are organised as follows:

- We analyse a commonly used SGX application benchmark, called *sgx-nbench* [6], and identify fundamental flaws.
- We implement a standardised and modular application benchmark framework for Intel SGX called SGXoMeter.
- We illustrate the performance degradation induced by the development progression of the SGX-SDK.

The rest of the paper is structured as follows. In § 2 we provide a brief introduction into SGX and the associated limitations followed

EuroSec'21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8337-0/21/04…$15.00
https://doi.org/10.1145/3447852.3458722

by some related work. Afterwards, we present *sgx-nbench* and discuss its flaws in § 3. In § 4, we introduce SGXoMeter followed by some evaluation results in § 5. Lastly, we elaborate more on future plans for the framework in § 6 and in § 7 we draw some conclusions.

2 BACKGROUND & RELATED WORK

In this section, we give an introduction into SGX and the bundled SDK from Intel including the accompanied restrictions and induced performance issues. Finally, we discuss some of the related work.

2.1 Software Guard Extensions

Intel has extended the x86 instruction set with new instructions to support its novel CPU feature Software Guard Extensions (SGX) [14, 27]. It adds a strong security property to sensitive code and data through secured compartments called *enclaves*.

Enclaves provide confidentiality and integrity protection with the support of the hardware. They are stored in a special section of the system memory called Enclave Page Cache (EPC) which is reserved by the processor at boot time. Currently, SGX limits the size of the EPC to 128 MiB, from which only ≈ 96 MiB can be utilised for enclave pages. Recently, the total EPC size was extended to 256 MiB with ≈ 188 MiB usable for enclave pages [2].

2.2 SGX Software Development Kit

Shortly after the release of SGX, Intel followed it with a Software Development Kit (SDK) [18] to ease the development of SGX-enabled applications. The SDK adds another level of abstraction of enclave transitions and introduces a new way of enclave interactions through *ecalls* and *ocalls*. Ecalls cross over the secure borders of an enclave from the untrusted part and vice versa for the ocalls. The resulting application is divided into two pieces, the trusted part containing the enclave's sensitive code, and the remainder of the program located in the untrusted part. The SDK also introduced the Un–/Trusted Runtime System (U/T–RTS), where the enclave transitions are handled. Ecalls and ocalls are described by the developers in an interface-like form using the Enclave Description Language (EDL). Later, the SDK's code generator, `sgx_edger8r`, generates wrapper code from this EDL file, which will be compiled and linked into the developed SGX application. Moreover, the SDK provides a trusted cryptography library, which contains basic encryption and decryption functions, and an SGX custom-made version of C/C++ standard library where functionalities that require unpermitted system calls are excluded. However, as a workaround, these functions can be outsourced by an appropriate ocall.

2.3 Limitations & Performance Issues

Reasonably, SGX's strong security guarantees inflict a performance overhead caused by multiple factors. First, enclave transitions, such as entering the trusted environment and executing code in it, are costly because of the security checks done at the enclaves boundaries or at the accessed memory buffers. As mentioned earlier, some system calls are not allowed inside the enclave. Therefore, an ocall needs to occur before desired system call, e.g., I/O operations [17] can be performed. This also impacts the performance especially when used frequently [11, 28].

The second major reason is enclave paging. Since the EPC size is limited, the SGX driver provides support for enclave paging from the EPC to the main memory to handle the oversized enclaves. Any enclave that exceeds the EPC's size limit would result in the previously mentioned paging process where enclave pages are encrypted before casting them out to the DRAM. However, this process has an overhead of up to three orders of magnitude depending on the memory access pattern [11, 12].

Considering the above limitations among others, SGX enabled applications should be built very carefully. Otherwise, it will result in impractical performance for products. Several research projects [11, 28, 33] proposed different mechanisms and concepts to mitigate the performance overhead of enclave transitions. However, these are either proprietary projects or require the application development to adapt to new programming paradigms [32].

2.4 Related Work

Some research papers executed micro-benchmarks to evaluate the previously stated factors for the performance overhead while utilising SGX. Weisse et al. [33] measured the overhead induced by the enclave transitions of the SDK's ecalls and ocalls and noticed an increase of about 8,600 to 14,000 cycles. Weichbrodt et al. [32] also ran overhead measurements for the same enclave transition, but in multiple settings scenarios and came up with similar results. In some cases, they applied *μcode* updates, which increased the security checks against side-channels attacks. Thus, they observed even higher overheads. Other papers, such as SCONE [11] and Secure-Keeper [12], have measured the impact of enclave paging on the performance. They observed an overhead of 5.5x upon exceeding the L3 cache size, and 1000x when hitting the EPC's size limit and above. All the aforementioned related works provide either only micro-benchmarks for specific scenarios, such as the enclave transitions performance overhead, or profiling and dynamic analysis tools of the executed SGX applications such as *sgx-perf* [32].

For research as well as industry SGX-related projects, it is important to have a standard application benchmark framework to measure the impact of the conducted changes into the SGX technology on the overall performance. In essence, it can be utilised after applying *μcode* patches to mitigate malicious attacks or updating SGX applications with the newest SDK version to measure the additional performance overhead costs. Moreover, due to the standardisation, this would allow researchers to compare and reproduce their results.

There are several research papers [15, 16, 30] that contributed mitigation mechanisms against some side- and controlled-channel attacks. However, the increase in security mostly incurs some performance loss. Therefore, Fu et al. [15] ported the ten benchmark programs from *nbench-byte* [1] into SGX to measure the overhead of their solution. They named it *sgx-nbench* and published it as an open source benchmark tool for SGX [6]. Due to the lack of application benchmark tools for SGX, *sgx-nbench* was used in other research papers [15, 16, 30] for their evaluations. Other researchers rather opted to port the *nbench-byte* benchmark programs themselves [30]. However, they did not provide the source code of their implementation, which leaves us only speculations and presumptions of committing similar flaws as the detected ones in *sgx-nbench*.

3 ANALYSIS OF SGX-NBENCH

In this section, we will take a closer look at *sgx-nbench* and explain its architecture and workflow. Then, we evaluate it based on two criteria, namely usability and its suitability as an SGX benchmark tool including each one of the ten ported programs separately. The correctness of the algorithm implementation of each benchmark program is orthogonal to this work. So, we assume that they were directly adopted from the original *nbench-byte* [1] tool. It is noteworthy that *nbench-byte* was developed in the mid-1990s. Hence, some of our evaluations are affected by this fact as well.

3.1 Design of sgx-nbench

The structure of *sgx-nbench* is composed of three components: I) The untrusted part, *app*, is where wrapper calls for all available ecalls are defined and an ocall for output purposes are implemented. It also contains the main function that does the enclave initialisation/destruction and executes the benchmark programs. II) The trusted part, *enclave*, consists of multiple source and header files that contain the SGX-port implementation of the ten benchmark programs, which are the following: 1-2) Numeric & string heap sort 3) Bit operations 4) Floating-point emulation 5) Signal processing using Fourier transformation 6) Assignment algorithm 7) Cryptographic operations using an international data encryption algorithm 8) Compression operation using Huffman 9) Backpropagation network simulation 10) Linear equations solving algorithm. III) The transition part, *nbench*, is where the tool's logic setup is located. It parses the input, sets up the global configurations, executes each benchmark program including pre/post-functions for initialisation and clean up purposes, and lastly processes the extracted results and generates statistics.

3.2 Workflow

The execution of *sgx-nbench* runs in three phases. In the first phase, it parses the input and allows to pass a configuration file. The user can use the latter to specify which tests to run and to adjust the configuration of the chosen programs, such as the size of the used memory buffers, number of loops or maximum time interval for each test iteration. In the second phase, it runs the benchmark programs in a fixed order. First, each program is executed five times and the results are stored in a fixed size array of maximum 30 entries. Then, it calculates the 95% confidence-half-interval of the generated results using the student-t-distribution with 29 degrees of freedom. As long as the value is not less than 1% of the results' mean value and the number of executions did not reach the limit (30x), then a new execution entry will be added to the result's array and the same calculations will be repeated. However, if one of the two above conditions is met, the mean value is stored for comparison and statistics purposes later in the third phase. Furthermore, in case the execution limit cap is reached while the first condition is still not met, then an error message regarding the inaccurate result of the executed benchmark program is returned. Thereafter, the same process is repeated for the subsequent benchmark programs.

By default, the first execution of each program does an adjustment step to initialise the used memory buffers with a predefined size inside the enclave and load them with pseudo random content. This pre-process is limited to a predefined number of cycles. Thus, the adjustment step is repeated with an incremented size of memory until it surpasses the fixed cycles rate to adaptively set the workload based on the CPU's performance. Afterwards, the actual benchmark process is executed via an ecall on the previously loaded data. This ecall is repeated in a loop fixed by a predefined time condition, meanwhile a counter is incremented for each iteration. Once the time condition is exceeded, the result is stored in form of the number of iterations per second. Eventually, a post-execution step takes place to free and clean up the used memory buffers.

The third and last phase is where the calculated statistics of each completed benchmark are outputted in a textual form in the console. By default, the mean value of the gathered results of each benchmark is calculated. Afterwards, it is compared to two hard-coded results of the same benchmark on two other different machines, DELL Pentium XP90 and AMD K6-233.

3.3 Evaluation

We summarise the evaluation of *sgx-nbench* in two sections starting with *usability*. The documentation is not sufficient which diminishes the usability as well as the comprehensibility of the tool. Furthermore, due to the missing usage documentation, one would run the benchmark with the default hard-coded configuration values. We noticed that some of them are rather impractical. For instance, the default cycle rate used for the adjustment step is set to 60 cycles which is not even sufficient for the enclave transition itself [32]. As a consequence, the adjustment step would always end up doing the same configurations. Thus, making it without a startup configuration file obsolete. Moreover, *sgx-nbench* does not have a warm-up phase. Although it is not really necessary for CPU heavy operations, we believe that heavy memory benchmarks could benefit from a warm-up phase to avoid outlier results caused by caching and to provide more accurate results since only a maximum of 30 samples are gathered for each single benchmark program. Another slight drawback we found was that the output results were not very intuitive due to the lack of documentation. The first column of the output shows the performance of the running machine (iterations/sec). However, the second and the third column show by which factor the current machine is faster than two other different ones. Moreover, the specifications of these machines are not completely provided and both ran on an outdated kernel version and an old C standard library.

We also analysed *sgx-nbench* regarding potential *flaws* and its *suitability* as an SGX benchmark tool. Aside from the cryptographic operations, compression and signal processing, the other seven benchmark programs are not critically security sensitive. Thus, despite their CPU intensive computations, they do not entirely fit as SGX-related benchmark programs. Furthermore, the loop step of the benchmark process measures not only the program's execution time but also the enclave transitions. Therefore, programs, like Bit-Flip, have sometimes shorter execution times than the enclave transition itself causing the generation of unreasonable results.

In summary, based on the aforementioned analysis and the current code quality (e.g. scattered hardcoded values and glue code) of *sgx-nbench*, we decided to implement the SGXoMeter framework. It addresses these issues and provides more accurate SGX-tailored benchmarking results.

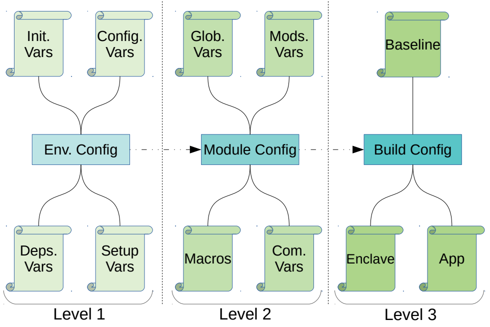

Figure 1: Hierarchy of SGXoMeter build structure

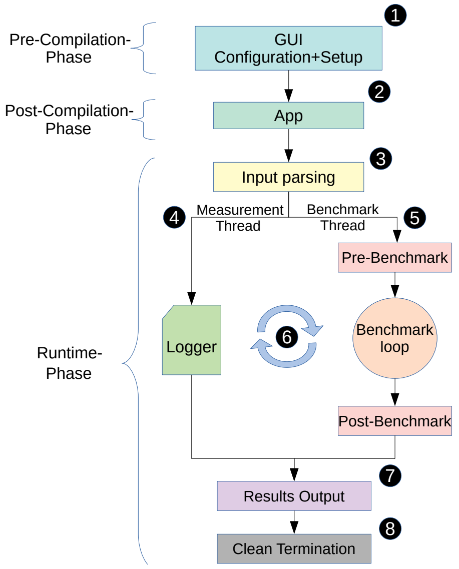

Figure 2: Architecture of SGXoMeter

4 THE SGXOMETER FRAMEWORK

In this section, we present *SGXoMeter*, an application benchmark framework for Intel SGX. It offers an abstract, modular and multi-threaded way of executing SGX application benchmarks while taking the detected problems mentioned in § 3.3 into consideration.

The *SGXoMeter* framework is developed using the C/C++ programming language and CMake as a build manager. The latter enabled us to build the framework in a modular way, such that every test application we implemented, also referred to as *module*, can be added or removed at compilation time. Thus, resulting in a variable sized enclave and several possible benchmark variants.

As can be seen in Figure 1, the hierarchy of *SGXoMeter*'s CMake structure is divided into three levels based on their location depth in the framework's file structure. The outermost space, level 1, is where the environment configurations are set up such as fetching SGX-SDK's libraries whereabouts, setting global compilation variables and installing needed dependencies like Intel SGX-SSL. Then follows the module configuration step in level 2. Common variables, macros and functions as well as important pre-compiler definitions for the default configuration values are defined there. The latter are visualised in the user-interface allowing selection or value modification. They also provide a brief overall description when hovering over them. Furthermore, default configuration values can be set at runtime using the command line arguments at the start. The build phase comes last at level 3, where the makefiles for the actual tool are generated based on the pre-configuration set in the earlier two stages. Eventually three executables can be built, a baseline and an SGX-Application, where the latter can be set either in hardware or simulation mode. These three variants execute the same chosen benchmark modules. However, the baseline only differs from the other two in running the test modules natively without SGX primitives. Therefore, this is the key condition to see how efficient utilising SGX in some sensitive procedures could be.

Figure 2 shows an overview of the framework's workflow. The first step ❶ is to setup the build folder, configure the tool, select the benchmark modules, and set the default values of the global variables. This can be done either using the graphical user interface or by manually editing the appropriate CMake files. Thereafter, the generated makefiles ❷ build the framework's executables, baseline and SGX-Application, with the chosen benchmark modules and the predefined global values. Upon launching either of the executables, both go through similar procedures. At the start, the passed command line arguments ❸ get parsed and a global configuration data structure is declared and defined with either the default or the passed values. Afterwards, an enclave is initialised and pointers to a shared counter and to the early mentioned global data structure are passed to it. Then, by default, two threads are created one for measurement ❹ and the other for running the benchmark modules ❺ inside the enclave. However, the framework also allows multiple in-enclave benchmark threads with separate counters to avoid synchronisation costs. For simplicity, we continue the illustration with a single benchmark thread. Upon creating the benchmark thread, the measurement thread will be paused until the pre-benchmark phase is finished. Thereafter, a signal from the main thread will trigger both threads, measurement and benchmark, to change their status to "running". At that moment, the benchmark thread starts its execution loop on the current module and increments the shared counter after each complete iteration. Meanwhile, the measurement thread runs in two stages, warm-up and real runtime, respectively. At the end of each stage, it logs the current value of the shared counter, number of iteration per second and the executed module's name. If the runtime stage is finished and there are still other modules to benchmark, the measurement thread signals the benchmark thread to set its status to "paused", leave its execution loop and enter the post-benchmark phase for cleaning up. Subsequently, the process repeats itself ❻ for the next module in the list. Finally, if all modules are successfully benchmarked, both threads are stopped and the main thread takes control back again. It harvests the previously logged results and either outsources ❼ them in a CSV format to an external file or outputs them back in the terminal. Eventually, the main thread ❽ destroys the enclave, cleans up used allocated memories and terminates the executable.

To avoid additional costs, the measured benchmark loop is kept minimal and it only contains the execution of the corresponding module's function and the incremented shared counter.

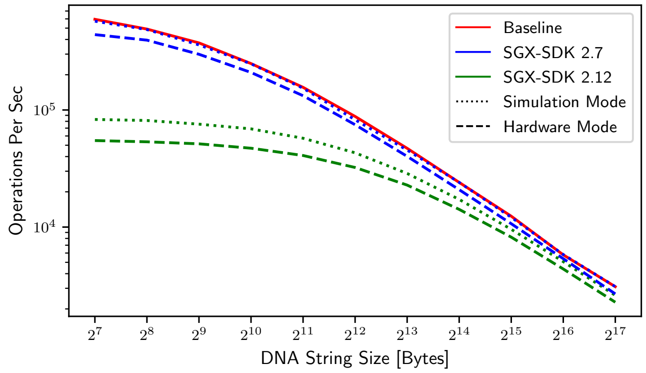

Figure 3: DNA pattern matching benchmark module

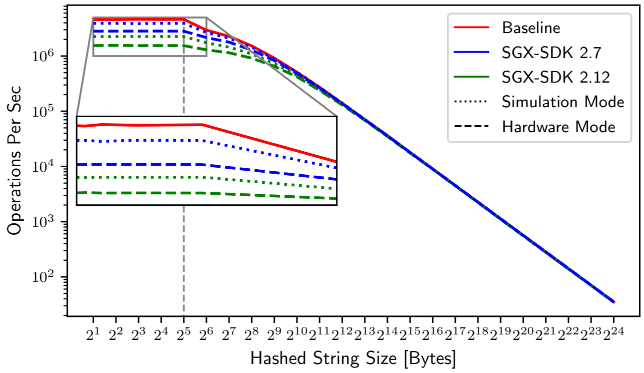

Figure 4: Hashing strings with Intel's SGX-SSL SHA256

We provide various benchmark modules which are mostly derived from the Intel SGX-SSL library [5] such as RSA and elliptic curve key generation, hashing using SHA256, en-decryption and signing/verification using RSA, and elliptic curve combined with diffie hellman and DSA. Furthermore, we analysed the source code of the Seeq library (see § 5) and implemented an SGX port for it and then add it as a module in the provided benchmark modules pool. To achieve this, we replaced all unsupported functions with SGX capable implementation such as I/O operations. Moreover, we modified the structure to reduce enclave transitions to a minimum.

Due to the modularity property of the framework, we also provided an interface-like file for custom benchmark module purposes. It enables third-parties to benchmark their SGX applications by implementing the three essential functions delivered with the interface. One is called in the pre-benchmark phase for the potential necessary preparations, the second executes the custom application and is placed inside the benchmark-loop, and the last one is called in the post-benchmark phase for releasing the reserved resources.

Lastly, we believe that such modules are good candidates as practical SGX test-applications considering their security sensitive nature as well as the complexity of the performed computations.

5 EVALUATION

In this section, we present our SGX-port of the DNA pattern matching library Seeq [4] followed by the results of the conducted benchmark on some of the implemented modules. SGXoMeter was successfully tested on three machines with different system specifications. The shown experiments results were gathered from a server machine that runs on Ubuntu 18.04 LTS with Linux 4.15.0-136-generic. Its hardware consists of an Intel Xeon E-2176G @ 3.70 GHz processor and 32 GB (@ 2666 MHz) of memory. All experiments were conducted with the same configuration 10s for warm-up and 60s for runtime. Our Baseline is running the same benchmark modules without SGX primitives.

Seeq [4] is an open source C library. It implements a DNA/RNA pattern matching algorithm that uses the *Levenshtein* distance metric [26] while performing sequence matching. As input, a file containing the DNA sequence alongside the DNA pattern are passed as application arguments. By default, Seeq searches the sequence for the given pattern and returns the number of each line where the pattern has been detected. However, Seeq also offers additional flags to support other options and configurations, such as setting the maximum Levenshtein distance, defining the behaviour when detecting a non-DNA character, format and misc flags. Furthermore, Seeq also provides functions for sequence extraction and trimming.

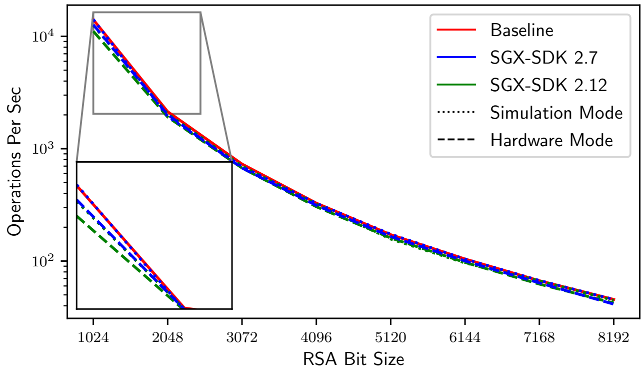

Figure 5: En-decryption using RSA keys of different bit sizes

Figure 3 shows the performance degradation between two different SGX-SDK versions (2.7 and 2.12) while running the DNA pattern matching benchmark on DNA sequences of different sizes searching for the same pattern. By default, *Seeq* prints the results out in the terminal but we omit them to avoid unnecessary enclave transition overhead costs. As can be seen, the performance of the new SDK releases degrade by nearly a factor of ≈ 10 compared to the baseline and ≈ 9 compared to an older SDK version. This performance overhead is most probably caused by Intel's mitigation mechanisms added to the SDK by each new release. Furthermore, the performance converges to more stable and comparable values the bigger the examined DNA size gets. This is because the pattern matching operation costs are significantly larger than the conducted defensive measures.

To make sure of this performance degradation, we ran other benchmark modules. For example, Figure 4 presents the results of running Intel's SGX-SSL port of the SHA256 on strings of different sizes. Due to hashing optimisations of short strings sizes, the performance starts to decay for sizes ($\geq 64 Bytes$). However, the same performance reduction pattern can be observed here as well.

The throughput of the newer SDK version has been decreased by a factor of ≈ 10 for small string sizes compared to baseline and by a factor of ≈ 5 compared to the older SDK version. For bigger string sizes, hashing gets more costly and the performance overhead differences become negligible.

We have also performed a variety of RSA benchmarks using Intel's SGX-SSL [5] implementation. For example, we benchmarked the en/decryption using RSA key pairs on strings with maximal possible sizes, which is the RSA's key byte size minus eleven. As shown in Figure 5, the impact of the newer SDK version is again only observable for small string sizes.

With the SGXoMeter framework, we are able to provide accurate and reproducible measurements in different system environment configurations. It allowed us to trace the discovered performance degradation back to the different SGX-SDK versions. More precisely, we repeated the benchmark process for the SGX-SDK versions between 2.7 and 2.12 with the same modules and identified an observable overhead jump starting with the 2.8 version. In addition, SGXoMeter targets standardisation and usage in a wider range of scenarios as we show in § 6.

6 FUTURE WORK

As a next step, we are planning to port *sgx-nbench*'s test programs in our framework and compare the generated results. Furthermore, we are considering to include an SGX port of the libsodium library [3] into our framework and add its various security sensitive cryptographic functionalities as benchmark modules. We also want to compare some hardware accelerated cryptographic mechanisms from different libraries, such as AES implementation in libsodium, SGX-SDK and Intel SGX-SSL. Sensitive OpenCV applications, such as face or fingerprint detection, are planned as well and we will port the necessary functions into SGX. Aside from adding new benchmark modules, we want to improve the usability of the code by implementing more features. For example, the ability to modify the enclave's configuration file by using the graphical user interface in the pre-compilation phase. In addition, provide a configuration option to include the enclave transitions, ecalls and ocalls, in the measurement. Moreover, add support to other SDK features like multiple enclaves and switchless enclave transition calls. Additional benchmark support to applications relying on other SGX frameworks like open-enclave [7] and Graphene-SGX [31] is of interest and will be investigated.

7 CONCLUSION

In this paper, we presented SGXoMeter to address the lack and necessity of standardised application benchmark tools for SGX applications. Due to the rapid development of Intel's SGX ecosystem and enhanced hardening against malicious controlled and side-channel attacks, the overall performance might vary substantially for several use-case scenarios. SGXoMeter is a modular multi-threaded application benchmark framework that measures the pure performance of SGX while excluding unnecessary costly enclave transitions. It offers a collection of suitable benchmark modules and allows the extension of custom ones through an interface that is easy to implement. Furthermore, it enables researchers to reproduce their results for various configurations, such as different *μcode* or SGX-SDK versions, and compare them against other related works. Finally, we were able to detect a performance degradation by a factor of up to ≈ 10× between two different SDK versions. The source code is available on GitHub[1].

ACKNOWLEDGEMENT

This project received funding from the German Research Foundation (DFG) under grant no. KA 3171/9-1 and KA 3171/12-1.

REFERENCES

[~~1~~] 1996. nbench-byte. https://www.math.utah.edu/~mayer/linux/bmark.html.

[2] 2000. 10th Generation Intel Core Processor Families Data Sheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf.

[3] 2013. A portable crypto library. https://github.com/jedisct1/libsodium.

[~~4~~] 2014. DNA/RNA pattern matching. https://github.com/ezorita/seeq.

[5] 2017. Intel®SGX-SSL. https://github.com/intel/intel-sgx-ssl.

[6] 2017. nbench-byte port for Intel SGX. https://github.com/utds3lab/sgx-nbench.

[7] 2017. Open Enclave SDK. https://openenclave.io/sdk/.

[8] 2019. Azure Confidential Computing. https://azure.microsoft.com/de-de/solutions/confidential-compute/.

[9] 2020. Confidential Computing Consortium Projects. https://confidentialcomputing.io/projects/.

[10] 2020. Signal Increases Their Reliance on SGX. https://medium.com/@maniacbolts/signal-increases-their-reliance-on-sgx-f46378f336d3.

[11] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*.

[12] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*.

[13] 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 *27th USENIX Security Symposium (USENIX Security 18)*.

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* (2016).

[~~15~~] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. 2017. Sgx-Lapd: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *Research in Attacks, Intrusions, and Defenses*, Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.).

[~~16~~] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. 2018. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. *CoRR* (2018).

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

[18] Intel. 2018. Intel Software Guard Extensions SDK for Linux. https://01.org/intel-software-guard-extensions/downloads.

[19] Intel. 2019. Intel Processor Microcode Package for Linux. https://github.com/intel/Intel-Linux-Processor-Microcode-Data-Files.

[~~20~~] Iulia Ion, Niharika Sachdeva, Ponnurangam Kumaraguru, and Srdjan Čapkun. 2011. Home is Safer Than the Cloud!: Privacy Concerns for Consumer Cloud Storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security (SOUPS '11)*.

[21] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper* (2016).

[~~22~~] David Kaplan. 2017. Protecting vm register state with sev-es. *White paper* (2017).

[23] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[24] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. 2015. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets-XIV)*.

[~~25~~] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy*.

[~~26~~] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*.

[1] https://github.com/ibr-ds/SGXoMeter

I am not having a literature review on a lot of the STX attacks specifically side channel attacks and their mitigations

[27] 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 '13)*.

[28] 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 '17)*.

[29] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[30] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.. In *NDSS*.

[31] 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)*.

[32] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. Sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference (Middleware '18)*.

[33] 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 '17)*.

[~~34~~] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*.