SGXTuner: Performance Enhancement of Intel SGX Applications via Stochastic Optimization

Giovanni Mazzeo, Sergei Arnautov, Christof Fetzer, and Luigi Romano

Abstract—Intel *SGX* has started to be widely adopted. Cloud providers (Microsoft Azure, IBM Cloud, Alibaba Cloud) are offering new solutions, implementing *data-in-use* protection via SGX. A major challenge faced by both academia and industry is providing transparent SGX support to legacy applications. The approach with the highest consensus is linking the target software with SGX-extended *libc* libraries. Unfortunately, the increased security entails a dramatic performance penalty, which is mainly due to the intrinsic overhead of context switches, and the limited size of protected memory. Performance optimization is non-trivial since it depends on key parameters whose manual tuning is a very long process. We present the architecture of an automated tool, called *SGXTuner*, which is able to find the best setting of SGX-extended *libc* library parameters, by iteratively adjusting such parameters based on continuous monitoring of performance data. The tool is — to a large extent — algorithm agnostic. We decided to base the current implementation on a particular type of stochastic optimization algorithm, specifically *Simulated Annealing*. A massive experimental campaign was conducted on a relevant case study. Three client-server applications — *Memcached*, *Redis*, and *Apache* — were compiled with SCONE's *sgx-musl* and tuned for best performance. Results demonstrate the effectiveness of *SGXTuner*.

Index Terms—Cloud Security, Intel SGX, Stochastic Optimization, Simulated Annealing

1 Introduction

The *Software Guard eXtension* of Intel's CPUs (SGX) is possibly the Trusted Execution (TE) paradigm which has the widest consensus of developers, software vendors, OEM, and software ecosystem partners. What makes SGX attractive is its capability of providing integrity and confidentiality protection of *data-in-use* even against super-privileged software and users. SGX is an application-layer Trusted Execution Environment (TEE) designed for security-aware developers, who can partition their application into processor-hardened enclaves or protected areas of execution in memory. The assumption is that only a small fraction of the application code runs in the TEE, which is called *secure enclave*. Consequently, the amount of protected memory that is made available to the secure enclave is small (128MB). Also notably, there are some functional limitations, and in particular it is not possible to issue system calls from within the secure enclave. Due to these limitations, the vast majority of legacy programs cannot use Intel SGX security features, unless (substantial) refactoring of the code is done. Given the tremendous potential of SGX, the research community — both in academia and industry — has engaged in a quest for solutions to overcome SGX design limitations, with the ultimate goal of making its superior security available to existing software in a transparent manner. *SCONE* [1], *Graphene-SGX* [2], *SGX-LKL* [3], *Haven* [4], *EleOS* [5], *Ryoan* [6] and *Panoply* [7] are remarkable examples of research endeavours that delivered effective solutions for transparent SGX support. These solutions were validated with respect to substantial case studies (e.g., Memcached, Redis, Apache, NGINX). We are also witnessing the release of the first commercial offerings of SGX-enabled cloud solutions for transparent *data-in-use* protection, notably: Microsoft Azure Confidential Computing [8], IBM Cloud [9], and Alibaba Cloud [10].

The approaches taken in the aforementioned solutions ultimately rely on some form of *system-level* support, that is, applications are loaded in the secure enclave and statically or dynamically linked with SGX-extended *POSIX* functions. By doing so, the gap between SGX-native and standard OS abstractions is bridged. Some differences exist, with respect to implementation aspects. A first degree of freedom is the specific *libc* library that is extended (e.g., *musl* [11], *glibc* [12], *eglibc* [6]). A second variant is the type of system call support (i.e., internal to SGX or external using dedicated *shields*). In all cases, the higher security comes at the cost of a dramatic performance penalty. This is primarily due to two intrinsic sources of overhead, specifically: *i*) the context switch between the secure and the insecure world, and *ii*) the paging activity when the size of the protected memory is exceeded. The performance penalty is even higher if the OS has been patched against the most recent types of side-channel attacks, namely *spectre* and *meltdown*. These attacks exploit a bug in the *speculative execution* of CPUs (i.e., a technique used by modern processors to speed up performance). The microcode update currently released by Intel to fix the bug disables speculative execution. As a consequence, it has a significant impact on performance (up to 25% [13] [14]).

Performance optimization of applications running on SGX is thus of paramount importance, but unfortunately it is a complex activity. The main difficulty is that the

- *G. Mazzeo and L. Romano are with the Department of Engineering, University of Naples 'Parthenope', Naples, IT. E-mail: {giovanni.mazzeo, luigi.romano}@uniparthenope.it*
- *S. Arnautov and C. Fetzer are with the Computer Science Department, Dresden University of Technology, Dresden, DE. Email: {sergei.arnautov, christof.fetzer}@tu-dresden.de*

Manuscript received May 29, 2018; revised April 5, 2019.

optimum is only achieved when all the key parameters of the SGX-extended *libc* libraries are properly set. The parameter set governs fundamental aspects with a direct impact on performance, including: compiler optimization, dynamic memory allocation, and synchronization between enclave and non-enclave threads. The number of possible configurations is vast. The evaluation of all combinations is so time consuming that manual optimization is not a viable option, since it could take years (even when it is performed by a domain expert). As an example, in our case study on SCONE [1], based on our detailed knowledge of internal mechanisms and the results of preliminary performance experiments (e.g., micro-benchmarks), we were able to simplify the optimization problem by defining boundaries and setting the step of variation of *Integer* parameters. Even so, we had to start from an initial problem space of 2.37×10^6 possible configurations, which was further reduced to 1.58×10^6 by stripping off combinations which we knew would have poor performance (again, based on detailed knowledge of internal mechanisms and results of preliminary experiments). Assuming 10 repeated tests with a duration of $20s$ each, the evaluation of the reduced space would still have taken ≈ 10 years of work, which is unacceptable. It is evident that an automated approach is the only option, and supporting tools are very much needed.

In this paper, we present the architecture and describe the operation of an automated tool, called *SGXTuner*, which can find the best setting of SGX-extended *libc* libraries, by iteratively adjusting parameters based on continuous monitoring of performance data. The tool is — to a large extent — algorithm agnostic. We decided to base the current implementation on a particular type of stochastic optimization, specifically the *Simulated Annealing* (SA) [15] algorithm. The main reason behind this decision is that SA — unlike other optimization algorithms such as hill climbing, genetic algorithms, gradient descent — has the advantage of avoiding being stuck in local optima, provided that the initial parameter values are properly set. In this work, the SA *temperature* and the related *cooling function*, controlling the convergence to the optimum, were chosen accordingly to Nourani et *al.* [16] based on a trade-off between convergence and searching time. To reduce the intrusiveness of the tool (i.e., to limit the impact of the tool on the target application's behavior), we developed *SGXTuner* as a lightweight distributed application. The current implementation of *SGXTuner* is written in *Rust lang* and is open source. It follows the microservice architectural pattern and is designed on a multi-container basis. It is composed of a *Core* microservice which houses the intelligence, and of a variable number of *Target* and *Benchmark* microservices executing — on separate physical nodes — the SGX application to be tuned and the benchmark tool, respectively.

SGXTuner was validated and evaluated with respect to a substantial case study, i.e., the extended version of the *musl* lightweight *libc* library, namely *sgx-musl* used in SCONE [1] and SGX-LKL [3]. Applications built with such a library may exhibit high performance variations, depending on the configuration of six key parameters.

To verify the effectiveness of our work, three widespread client-server applications built with *sgx-musl* were tested: two in-memory storage systems, i.e., *Memcached* and *Redis*, which are multi-threaded and single-threaded, respectively, and the *HTTP* multi-threaded *Apache* web server. Different versions of SA were compared to demonstrate the flexibility of the tool with respect to the specific algorithm chosen. We compared the sequential SA with three parallel implementations of SA. The benefit from parallel SA is twofold: *i*) it drastically cuts the searching time; *ii*) it allows the adoption of more extreme SA settings that could increase the probability of reaching the global optimum (e.g., high temperature and slow cooling function) while maintaining a reasonable searching time.

Results of the experimental campaign demonstrate the effectiveness of the proposed approach. There is a substantial improvement of throughput-latency for *Memcached*, which can vary from $\approx$ 11-12% respectively (if optimization is done starting from an initial pre-optimized configuration, defined by a domain expert) up to $\approx$ 45-39% (if the initial configuration is chosen in a random fashion). Similarly, *Apache* reported a performance improvement, which ranged from $\approx$ 14-37% up to $\approx$ 38-51%. Regarding *Redis*, instead, the gain is slighter (from $\approx$ 7%-5% to $\approx$ 13%-10%). The margin for improvement of single-threaded applications is narrower.

Overall, this paper provides the following contributions to the wide community of SGX architects and users:

- First, it proposes a general method for performance enhancement of applications secured with one of the most prominent techniques for providing transparent SGX support.
- Second, it shows the applicability of the solution in a representative case study, i.e., *sgx-musl* at the base of *SCONE* and *SGX-LKL*.
- Last but not least, it provides recommendations on how to define initial settings when single- or multi-threaded applications have to be secured. It does so by identifying weights and specific combinations of parameters with relevant impact on performance, using the *SGXTuner* datasets of optimal solutions.

The remainder of this work is organized as follows. Section 2 provides background on Intel SGX. Section 3 overviews current approaches of SGX transparent support for legacy applications and the parameters that could affect the overall performance. Section 4 states both problem and goals. Section 5 describes the design of *SGXTuner*. Then, Section 6 overviews the selected Simulated Annealing stochastic algorithms used in *SGXTuner*. Section 7 reports the conducted experimental evaluation. Section 8 gives an insight into related work. Finally, Section 9 concludes the document.

2 Background on Intel SGX

The 7th generation of Intel's CPUs is equipped with an innovative secure extension to the Instruction Set Architecture (ISA), namely Software Guard eXtension (SGX) [17]: a TEE based on a mechanism of "reverse sandbox" in which sensitive processes' address space is protected — at the CPU level — from the OS. The idea behind is to protect selected code and data from disclosure or

modification through the use of secure enclaves, i.e., address regions whose content is protected — via encryption and hashing — from any software outside the enclave, included privileged ones. Only the enclave code can access any part of the address space, except those areas belonging to other enclaves. The boundary between enclave and non-enclave sections is governed by the processor that blocks any access attempt from unauthorized processes. An interface — defined in a domain-specific C language — is declared by the programmer to establish entry points, i.e., calls to/from an enclave (namely ECALLS and OCALLS).

The strong security guarantees ensured by SGX do not come free of costs. First, the performance overhead is non-negligible. In particular, the execution time highly suffers from the context switch between enclave and non-enclave areas. This includes several computational intensive operation such as: monitoring the SGX boundary to prevent enclave memory from being read or modified by non-enclave code within the CPU (in SRAM), saving/restoring thread context, and invalidating the hardware Translation Look-aside Buffer. Furthermore, the memory encryption and integrity verification mechanisms made by SGX increase the probability of last level cache misses while reading enclave pages. This could entail an additional overhead on applications' performance.

Second, the physical memory that stores the Enclave Page Cache (EPC) — i.e., the data structure containing the protected code and data — is limited to the size of the Processor Reserved Memory (PRM) (128MB) instantiated by the BIOS at boot-time. Nevertheless, the PRM size limit can be extended (via software) up to 4GB on Linux OS since its driver supports paging. Hence, the hardware PRM size is always 128MB and the OS swaps the different pages of the protected enclave memory. This highly impacts the overall performance due to the costly encryption and decryption of enclave pages to be moved in untrusted memory.

Third, system calls instructions are forbidden inside the enclave. In fact, with Intel SGX, a compromised operating system is within the threat model. The execution of a system call needs to be enabled by the SGX developer that implements dedicated OCALLs for this purpose. In case of I/O-intensive applications, this SGX limitation causes a dramatic performance penalty since the program flow is continuously moved in and out of the enclave entailing the execution of several context switches.

3 Transparent SGX Support: Solutions and Tunable Parameters

The adoption of SGX *as-is* imposes to security engineers a dedicated partitioning of applications in secure and non-secure sections whose crossing functions must be defined and regulated via the enclave interface. There is always a trade-off to be faced between what to put inside and what outside. In this typical situation the developer needs the source code of the software to be secured and a proper knowledge of its architecture for the specific code refactoring. The transparent provision of SGX security features to legacy applications is made difficult by SGX's design principles and limitations, i.e., the limited amount of protected memory resources available to SGX and the impossibility of issuing system calls from the *secure enclave*.

The research community, first, and the industrial sector, later, have proposed solutions enabling the usage of Intel SGX for large and complex workloads — such as enterprise-level services or even public cloud applications — in a transparent manner. In this sense, *SCONE* [1], *Graphene-SGX* [2], *SGX-LKL* [3], *Haven* [4], *EleOS* [5], *Panoply* [7], *Ryoan* [6] represent significant research solutions, whose validation was realized via well-known web services (e.g., Memcached, Redis, Apache, NGINX). At the same time, first examples of commercial products have been recently released by Microsoft with its Azure Confidential Computing [8] that transparently secure data processing in the cloud, or by IBM Cloud [9] and Alibaba Cloud [10] that just recently announced the adoption of the Fortanix Runtime Encryption [18] based on SGX for *data-in-use* shielding. Clearly, we cannot be sure about architectural choices of commercial products as their details are unknown to the public. However, we can imagine that research paved the way to industrial solutions as in the case of Microsoft, whose research department published *Haven* [4], which provides *system-level* SGX support to unmodified legacy Windows applications. In all cases, the approach pursued for transparent SGX security is essentially the same and consists in providing a *system-level* support [19] (Figure 1). That is, applications are loaded in the secure enclave and statically or dynamically linked with SGX-extended *libc* libraries (i.e., the core standard C libraries of Linux systems) to perform *system calls* (i.e., the service requests made to the operating system kernel). However, the current solutions present some differences on the implementation side.

First, the extended *libc* library. SCONE and SGX-LKL use the lightweight *musl* [11], Graphene-SGX adopts *glibc* [12], EleOS leverages *glibc* as well, Ryoan uses *eglibc*, Panoply excludes *libc* from its TCB, to fit into the range of automated formal verification, as they shield at the *libc* interface.

Second, the type of system call support, i.e., *i*) running shielded external *syscalls* totally outside the SGX enclave via the SGX-extended *libc*, like in the case of *SCONE*, *Ryoan*, *EleOS*, and *Panoply ii*) or executing a subset of *syscalls* inside via a *Library OS* and keep issuing the unsupported ones externally via the SGX-extended *libc* library. This approach is pursued, e.g., by *Graphene-SGX*, *SGX-LKL*, *Haven*.

Regardless of these differences, most of the solutions (*SCONE, Panoply, EleOS, Graphene-SGX, SGX-LKL*) share a similar approach to reduce the SGX context switches when system call must be issued. That is, they enforce a M:N threading model in which M application threads inside the enclave are mapped to N OS threads. Several threads running outside in non-enclave mode are maintained to relay *syscalls* on behalf of enclave threads. In this way, the enclave thread initiating the *syscall* does not have to leave the enclave. Enclave and *syscall* threads typically use queues or buffers to communicate. An enclave thread adds a system call to the request queue. System call threads periodically poll the queue and execute system calls if there are outstanding requests. Results of system calls are pushed to the response queue and consumed by enclave threads. Depending on the *synchronous* or *asynchronous* mode, while a system call is being executed by the OS, an enclave thread could switch to another application thread to avoid busy

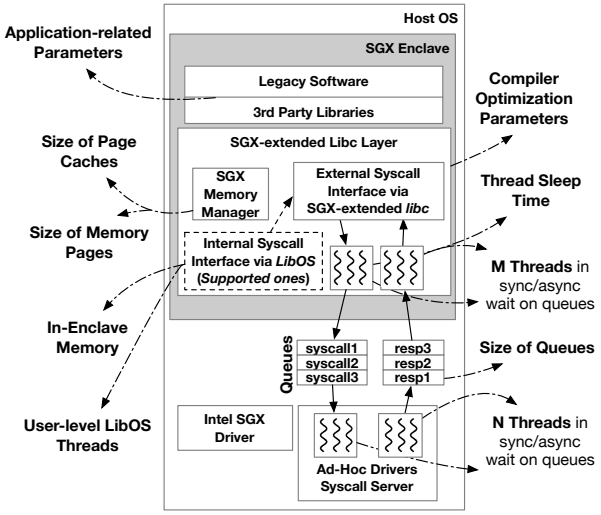


Fig. 1: Parameters affecting performance of SGX applications

waiting for the result.

As reported in Figure 1, there are many parameters — configurable at compilation- or run- time — whose impact on the overall performance of the SGX-enabled application can be significant. Some of these are independent on the chosen solution. In this sense, there are often parameters to be set on the specific legacy software to be secured. For example, *NGINX* and *MySQL* allow the configuration of buffer/cache sizes that — if properly set — could avoid the heavy SGX memory paging.

NGINX Parameters

```
client_body_buffer_size
client_header_buffer_size
client_max_body_size
large_client_header_buffers
open_file_cache
open_file_cache_valid
```

MySQL Parameters

```
innodb_buffer_pool_size
innodb_log_file_size
innodb_log_buffer_size
innodb_file_per_table
query_cache_size
```

Other parameters regulate optimization options of the specific compiler, which could help to reduce code size and execution time at the cost of slower building time.

Compiler Optimization Parameters

```
-O      -O2      -O0
-O1     -O3      -Os
```

There are then parameters, which have slightly different effects on: *i*) the memory management of SGX enclave pages, *ii*) the number of threads, the sleep time, and the queue sizes of the M:N threading model. In the rest of this section, we provide more details on specific transparent SGX solutions and their parameters. We first report details on our case study, *SCONE*. Then, we describe some additional example of works whose implementation details are available to the public.

3.1 SCONE Parameters

SCONE [1] proposes a SGX-enabled secure container solution to protect sensitive unmodified applications. The idea behind SCONE is to expose a C standard library interface to container processes, which is implemented by statically linking against a *libc* library within the SGX enclave. System calls are executed outside the enclave, but they are shielded by transparently encrypting/decrypting application data: files stored outside the enclave are therefore encrypted, and network communication is protected by transport layer security (TLS).

At the core of SCONE there is the convenient C standard library with SGX support, namely *sgx-musl*. Instead of leaving the enclave to perform a system call — *sgx-musl*, as described in [1] — supports a synchronous and asynchronous system call mechanism based on the M:N model of message exchange between the code running inside and outside enclaves.

In this regard, *sgx-musl* comes with six parameters configurable at run-time via as many environment variables, which can highly affect the overall performance of applications compiled with the SGX-aware C library. SCONE allows the user to specify: *i*) `ETHREADS` and `STHREADS`, i.e., the number of OS threads that run inside of the enclave as well as the number of system call threads that run outside; *ii*) `ESPINS` and `SSPINS`, i.e., the number of attempts that threads make to dequeue an element from request or response queues; *iii*) `ESLEEP` and `SSLEEP`, i.e., the sleep time of enclave and non-enclave threads when the specified number of attempts has been made and the thread did not succeed to dequeue an element and thus it enters in a sleep condition to prevent wasting CPU cycles.

SCONE Parameters

```
ETHREADS    ESPINS    ESLEEP
STHREADS    SSPINS    SSLEEP
```

3.2 SGX-LKL Parameters

SGX-LKL [3] is an additional solution for transparent SGX support, whose approach is to execute *syscalls* directly inside the SGX enclave via a Library OS (*LibOS*). That is, a new paradigm trend where kernel functions are available to user space (*ring3*) programs in a form of a library. SGX-LKL leverages a particular implementation of a *LibOS*, namely *Linux Kernel Library (LKL)* [20], which allows to port all *syscalls* in the enclave except for I/O-related ones. For this kind of kernel functions, instead, *SGX-LKL* makes requests to the outside world via *sgx-musl*. In practice, *syscalls* are carried out within the enclave using LKL only when possible, otherwise the same M:N asynchronous system call mechanism adopted in SCONE is used for the subset of system calls that require direct access to external resources and are therefore processed by the host OS. The main advantage of this approach is the reduced exposure of data to the untrusted world, although this comes at the cost of a larger TCB size.

Even in this case, there are parameters affecting the overall performance of the SGX-secured application. Some of them correspond to the *sgx-musl* parameters we saw before, which are also configurable at run-time. Other settings, instead, are specific of the *SGX-LKL* library: *i*) `SGXLKL_MAX_USER_THREADS` to define the maximum number of LKL user-level threads inside the enclave; *ii*) `SGXLKL_HEAP` to set the total heap size available in the

enclave; *iii*) SGXLKL_STACK_SIZE to assign the stack size of in-enclave and user-level LKL threads; *iv*) and SGXLKL_SHMEM_SIZE to define the size of the shared memory to be used between the enclave and the outside world.

SGX-LKL Parameters

SGXLKL_ETHREADS	SGXLKL_SSPINS
SGXLKL_STHREADS	SGXLKL_ESLEEP
SGXLKL_ESPINS	SGXLKL_SSLEEP
SGXLKL_MAX_USER_THREADS	SGXLKL_HEAP
SGXLKL_REAL_TIME_PRIO	SGXLKL_STACK_SIZE
SGXLKL_SHMEM_SIZE	

3.3 EleOS Parameters

EleOS [5] is a runtime system that aims at transparently providing OS services in the enclave without exiting the TEE. In this solution, *syscalls* and user managed virtual memory — namely Secure User-managed Virtual Memory (SUVM) — are performed in the enclave. As for previous approaches, an important goal is to reduce the overhead due to the SGX context switches. System calls are delegated to a remote procedure call (RPC) service running in another application thread, without exiting the enclave. An M:N threading model is used to decouple threads internal and external to the enclave. The virtual memory, instead, is managed in the enclave by equipping SGX with a software-based paging system for C++ based programs. A key abstraction of the *EleOS* design are *spointers*, a specific instance of smart pointers that can determine if referenced data is inside or outside the EPC. As a consequence data can be paged into the enclave without mode transitions. The idea is that "users allocate buffers in SUVM via a special allocator and obtains the special *spointer*, which can then be used as a regular pointer in the application". The *spointer* accessing evicted SUVM pages triggers a software page fault, which is handled entirely inside the enclave.
EleOS is highly configurable. Currently, configurations are determined at compile-time. Authors claim that support for runtime configurations will come soon. Parameters can be tuned for both components implemented in *EleOS*, i.e., the RPC system and the SUVM mechanism. A boolean parameter establishes which ocalls use the RPC system. Like in SCONE, the RPC threads communicate with the enclave threads through queues. Integer parameters are used to control the queue size (QUEUE_SIZE) and the number of active RPC threads (EXTERNAL_THREADS, INTERNAL_THREADS), together with their sleep time (SLEEP_TIME).

EleOS Parameters

QUEUE_SIZE	PAGE_CACHE
EXTERNAL_THREADS	PAGE_CACHE_SIZE
INTERNAL_THREADS	PAGE_SIZE
SLEEP_TIME	

Moreover, an integer parameter can control the size of pages allocated in memory (PAGE_SIZE). Another integer parameter defines the page cache size (PAGE_CACHE_SIZE). The "backing store" (i.e., the untrusted memory) size is also tunable through an integer setting. Finally, a boolean parameter tells to *EleOS* if the page cache (PAGE_CACHE) has to be used, thus enabling the so-called "direct access to the backing store".

4 Problem and Goal Statement

This paper covers the following *parameter tuning* problem. Given:

- A legacy *target* application A, SGX-enabled with one of the *system-level* solutions presented before.
- Parameters $p_1, ..., p_n$ configurable at either compilation- or run- time, whose typology varies depending on their set of values, namely: *i*) *Categorical*, i.e., parameters having a finite, unordered set of discrete values; *ii*) *Boolean*, i.e., parameters with only two possible logic values; *iii*) *Integer*, i.e., parameters whose domains of values are discrete and ordered.
- A space C of configurations (or *parameter settings*), where each configuration $c \in C$ specifies values for A's parameters such that A's behavior on a given problem instance is specified.
- A set of problem instances I
- A performance metric m measuring the performance of A, on instance set I for a given configuration c.

The goal can be stated as follows: find in a limited amount of time a configuration $c^* \in C$ that results in optimal performance of A on I according to metric m. The number of parameters and their typology determine the nature of the configuration space C and have profound implications on the approach to pursue.

The majority of parameters seen in section 3 are of *Integer* type, thus their space C is extremely vast. However, relying on an extensive domain expertise could significantly reduce the space of configurations. The user could make assumptions on the type of parameters (e.g., *Integer*, *Boolean*) and their values. This could result in significant performance improvements. Preliminary experiments could be also executed to make preliminary estimates. For example, in the case of SCONE and its *sgx-musl*, our domain knowledge and initial evaluations led us to define values' boundaries and step of variation reported in Table 1. The following considerations were made.

If the *target* is a multi-threaded application, the number of ETHREADS must be near the number of processor cores, while for single-threaded software $\leq$ 2. Regarding STHREADS, it must be larger than ETHREADS to ensure a service rate higher than the arrival rate on queues. We observed that it also depends on the number of *target* application's threads. For each application thread it is required at least one STHREAD, otherwise the *target* application could lock up. This could happen if all available STHREADS are performing blocking *syscalls*, and an application thread that does not have a corresponding STHREAD would not be able to make progress until some of them are unblocked. Having more STHREADS than the number of *target* application's threads allows to shorten the time between a *syscall* being added to the *syscall* queue and consumed by an THREADS. On the other hand, having too many STHREADS could lead to very high contention on the queues, thus reducing application performance. In general, we noticed that

values greater than 20 and lower than 70 seemed to produce better performance. Then, setting SSPINS and ESPINS to low values in the order of 10 and 1000, respectively, would drastically reduce the performance while too high values would force threads to busy-wait for system call requests and responses, thus increasing the usage of system resources. We also discovered that enclave threads should have sleep times (ESLEEP) substantially higher than system call threads (SSLEEP) otherwise the overhead on I/O operations diverges.

Under such assumptions the space C reaches 2.37×10^6 configurations. From such a space, we take out the worst ones that would make no sense to evaluate. As an example, we noticed from preliminary evaluations that combinations of parameters used in multi-threaded applications where ESPINS=3400 and ETHREADS=1 highly increase the overhead in terms of latency within the enclave. At the end, we created a subspace C_s composed by 1.58×10^6 configurations.

The majority of *target* applications (A) to be secured with SGX are client-server web services, whose throughput and latency (i.e., metrics m) are typically evaluated with benchmarks that for each run (i.e., instance I) take on average $20s$. Considering that at least 10 instances I must be repeated for the same configuration C to produce reliable results, the time required to evaluate the entire subspace C_s would be ≈ 10 *years*.

Parameters	Lower Bound	Upper Bound	Step of Variation
STHREADS	20	70	5
ETHREADS	1	10	1
SSLEEP	2,8k	4,2k	100
ESLEEP	12k	20k	1k
SSPINS	50	140	10
ESPINS	3,4k	6,4k	200

TABLE 1: Configuration Space of *sgx-musl*

Hence, even in the best conditions — i.e., considering the sole *sgx-musl* related parameters, making initial assumptions based on domain knowledge, and reducing the initial space C — the time required to evaluate C_s is unpractical.

5 *SGXTuner* Design

We propose the *SGXTuner* approach to find optimal configurations in an extremely low time. *SGXTuner* does this by taking advantage of *Stochastic Optimization (SO)*, i.e., a method for minimizing or maximizing an objective function in the presence of randomness. SO is widely used in a variety of disciplines where finding precise problem solutions through deterministic methods would be practically impossible. In the rest of this section, we describe the design of *SGXTuner*. We present the architecture and the typical sequence of actions during a tuning activity. Then, in the following section, we introduce the particular stochastic algorithm adopted in our solution.

SGXTuner is a tool for automatically tuning *system-level* solutions supporting SGX-enabled applications. It is able to search for an optimal configuration in either sequential or parallel execution. The tool is — to a large extent — algorithm agnostic. Figure 2 shows the architecture of *SGXTuner* with its main components organized in three layers: *i*) the *Control* hosting the intelligence of the system; *ii*) the *Application*, which comprises the actual *Target* application to be evaluated and the related *Workload* generator; *iii*) the *Infrastructure* layer including the physical nodes of the infrastructure used for the deployment of testing units.

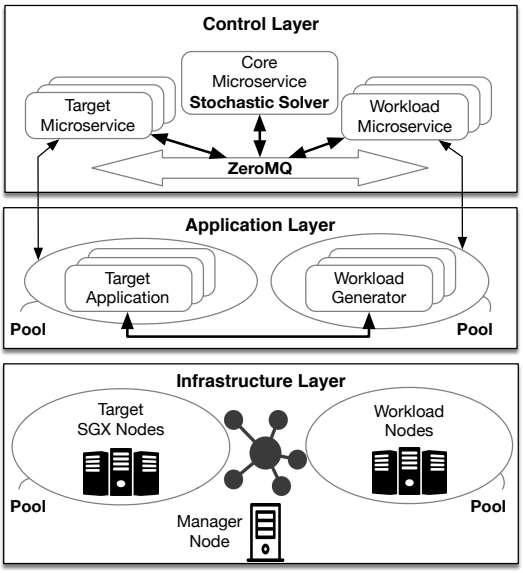


Fig. 2: *SGXTuner* Architecture

The *Control* layer is where decisions are taken about the specific job, and in particular on: *i*) configurations of parameters to be tested, *ii*) the assignment of a selected configuration to a *Target* application to be executed in a specific SGX Node taken from the pool, *iii*) the dispatch of a *Workload* generator to a specific node for the execution against the previously-chosen SGX node. From an implementation point of view, the *Control* layer is based on microservices, which exchange control messages through the *ZeroMQ* (ZMQ) channel, a high-performance asynchronous messaging library used in distributed or concurrent applications. The *Control* layer includes three type of microservices. The *Core* microservice runs the stochastic algorithm and decide the parameters to evaluate. This microservice keeps track of the applications' status in two pools and assigns jobs to the first available ones. The *Target* microservice receives from *Core* the values of the configuration parameters, their type (e.g., integer, boolean, string), and the configuration level (i.e., compile- or run-time). These are then used to launch the *Target* application. Finally, the *Workload* microservice receives from *Core* the IP address of the chosen *Target* application, the metric of interest to be returned (i.e., latency or throughput), and other information regarding the specific benchmark tool to be used. At the end of each run, results are returned to the *core* microservice, which will use them within the stochastic algorithm for taking decisions on the next configuration parameters to evaluate.

The *Application* layer, instead, encloses the applications driven by the *Control* layer to perform the tuning activity. More precisely, these are: *i*) the sensitive *Target* application

(e.g., NGINX, Apache, Memcached, or MySQL) — hardened with the particular *system-level* SGX solution — whose performance needs to be improved, *ii*) and the *Workload* generator such as a benchmarking tool (e.g., wrk, memaslap, or twemperf) that is chosen based on the *target* application.

Finally, the *Infrastructure* layer is used for the deployment of the *Target* and *Workload* applications on the physical machine nodes. A first category of machines must include the SGX hardware extension. A second category is needed for the execution of the *Workload* generator and does not require SGX. An additional machine, the *Manager* node, may be used for the execution of the core microservice to avoid that performance results are altered when the stochastic algorithm is too much intrusive. In order to increase the efficiency of *SGXTuner*, both *Target* and *Workload* nodes are organized in pools. It must be noticed that a one-to-one correspondence between nodes and applications should be preferred. In case of a reduced number of machines, it may be possible to execute both the *Target* and *Workload* applications on a single node. Or even, deploying the stochastic solver and the *Workload* generator on one powerful machine. In short, there could be degrees of freedom for what concerns the allocation of application's units. However, it is important to avoid situations causing interference in the performance measurements, which could lead to wrong application settings.

6 Stochastic Optimization: Simulated Annealing

Among the available stochastic optimization algorithms — such as hill climbing, genetic algorithms, or gradient descent — in our *SGXTuner* prototype we used *Simulated Annealing* (SA). This has the strength of avoiding to stuck in local optima, i.e., solutions that are better than any others nearby, but are not the very best. SA was proposed by Van Laarhoven et *al.* [15] as a simulation of the metallurgical annealing process. It finds the optimal global *Energy* by gradually decreasing a parameter named *Temperature*. Initially, the temperature is high, and several transitions occur for both low-energy and high-energy solutions. This makes possible the identification of the global optimum by searching a wide solution space. As the temperature cools, the frequency of transitions to worse states decreases, and the system tends to move in better states more frequently. It is important to notice that the temperature must go down at a reasonable rate allowing the method to search thoroughly at each temperature.
In the following, we report fundamentals of SA, and then also explore parallel implementations of such stochastic algorithm.

6.1 Fundamentals and Settings of Simulated Annealing

The SA tuning algorithm can be defined as a 6-tuple $\langle \mathcal{A}, S, \Omega, T, \Psi, \Phi \rangle$, where $\mathcal{A}$ represents the target application, S the parameters state space, Ω the objective function, T the initial process *Temperature*, Ψ the *Cooling Strategy*, and Φ the *Termination Criterion*.
At the beginning, SA evaluates an initial predefined state $s_0 \in S$ to determine the first candidate solution with an associated *Energy* $E(s)$. Then, the stochastic process iterates as follows:

(1) A new state s' — extracted from the set of possible *Neighborhoods* $\mathcal{N}(s') \subset S$ — is evaluated. The resultant energy $E'(s')$ is then used to calculate the Δ difference with the current selected best energy $E(s)$: $\Delta E = E(s) - E'(s')$
(2) If the objective function is a maximum function evaluation ($max(\Omega)$), the transition from s to s', at step k, is done if the *Boltzmann Criterion* is verified:

$$R(0,1) \leq \begin{cases} 1, & \text{if } \Delta E_k \leq 0 \\ e^{-\Delta E_k / T_k}, & \text{otherwise} \end{cases} \quad (1)$$

Where $R(0,1)$ is a random number in $[0,1]$, and $e^{-\Delta E/T}$ is the *Metropolis* function. The probability of moving from one state to another depends on: 1) ΔE, i.e., when new states with better energies are encountered; 2) T, in fact if T tends to zero, the probability R also tends to zero. Hence, the algorithm allows the possibility that even worse solutions can be accepted. This ensures the search process does not hang in local optima.
(3) Temperature T is decreased using a predetermined Ψ cooling schedule $\rightarrow T_{k+1} = \Psi(T_k)$
(4) The termination criterion Φ is checked. If this is not satisfied, the algorithm keeps iterating

Some settings of SA need to be defined *a priori*. The convergence time and the goodness of the final result highly depend from such a configuration. In this work, the following choices have been taken:

Initial Temperature (T_0) - The definition of T_0 is fundamental as, on the one hand, setting too high values could mean a slow convergence to the optimum. On the other hand, the usage of too low values implies the risk of being stuck in local optima. A general solution does not exist. It highly depends on the problem under study. In this work, the algorithm has been executed for a limited number of iterations to estimate the average ΔE. Afterwards, T was set such that the acceptance ratio of bad moves is equal to:

$$e^{-\Delta E/T} = 0.98 \Longrightarrow T = \frac{\Delta E}{-ln(0.98)} \quad (2)$$

In this way, most of the solutions are accepted during the first steps of the tuning process.

Cooling Strategy - Even the choice on the cooling strategy is important. Nourani et *al.* [16] report and analyse a wide set of annealing schedule. Lowering T in the wrong way could mean that SA does not reach the global optimum state but freezes into one of the local optima. A particular form of an *exponential* cooling strategy is used in this work:

$$T' = T_0 e^{-ln(\frac{T_0}{T_n})\frac{k}{k_{max}}} \quad (3)$$

The temperature decrease is computed using the ratio of initial (T_0) and (T_n) final temperatures, the algorithm progress status evaluated through the ratio of current SA step k, and the maximum number of steps k_{max}.

Termination Criterion At a certain point, the algorithm needs to determine a stopping condition. The SA process

terminates either if a convergence to an optimum is reached or if the number of steps performed is equal to the predetermined maximum steps k_{max}. In this paper, the upper bound of iterations is set to a high value, that is, $k_{max} = 10000$. The movement of solution, instead, is used as a convergence condition. If the objective function does not improve after 3% steps of k_{max}, then, the optimum is assumed to be reached.

6.2 Parallel Simulated Annealing

SA is inherently sequential, thus leading to long computation time, especially in case the stochastic algorithm is applied to problems with large search spaces, with high temperatures, and slow cooling strategies. The efficient way to speed up the SA algorithm, and make it more attractive for the optimization problem covered in this work, is to add parallelism. The usage of parallel SA could also enable the adoption of algorithm settings, which could increase the probability that the global optimum gets reached (i.e., high temperature and slow cooling function) while maintaining a reasonable searching time. It is therefore of interest to evaluate the impact of parallel SA algorithms in our *SGXTuner*, thus evaluating the optimum they produce.

Several parallel SA algorithms have been proposed over the years [21] [22] [23]. According to the classification made by Greening et *al.* [24], there exist three categories of parallel SA: 1) *Serial-like* algorithms that maintain the same characteristics of sequential ones [22]; 2) *Altered* algorithms [21] [23], which modify the procedure by, e.g., changing the state generation. 3) *Asynchronous* algorithms that try to reduce idle time and communication by calculating costs with outdated information. One of the most accepted example of *altered* algorithms is the *Parallel Recombinative Simulated Annealing (PRSA)* proposed by Mahfoud et *al.* [21]. The rationale behind PRSA is merging parallelism and convergence properties of *Genetic Algorithms (GA)* [25] and *simulated annealing*, respectively. The hybrid PRSA iteratively generates and analyzes a population of individuals (or states) of a certain size. The population evaluation consists in extracting two individuals (known as *parents*) and applying genetic recombination (*crossover*) and *mutation* to create two new individuals (*children*). New populations are constructed by selection from old and new individuals. The selection strategy between *parents* and *children* states is based on the SA Boltzmann. In contrast to normal genetic algorithms, therefore, the selection process is controlled by the SA *temperature*. Selections are mostly random when the temperature is high; then, when the temperature starts reducing, the selection of an individual depends increasingly on its fitness just as in a GA.

Olenšek et *al.* [23] investigate an asynchronous parallel SA algorithm. They propose a hybrid method that combines SA and *Differential Evolution (DE)*. "The random sampling and the Metropolis criterion from SA are combined with the population of points and the sampling mechanism from DE to balance global and local search". The new method is called *Parallel Simulated Annealing Differential Evolution (PSADE)*. "To reduce the optimization time, the method is designed as an asynchronous master-slave parallel system". An example of a *Serial-like* algorithm is the *Multiple Independent Run (MIR)* proposed by Lee et *al.* [22]. Authors propose a trivial but, at the same time, effective solution, i.e., running parallel independent SA with their initial state and temperature. Therefore, *workers* will not need to communicate anymore for state moves or solutions information. The best solution found is then chosen as the final result. The rationale behind MIR is that the goodness of the final solution often highly depends on the initial state. Hence, starting from many different states increases the probability of obtaining a result as close as possible to the optimum.

In this work, we want to demonstrate that even time-efficient SA algorithms using parallelism can be leveraged to optimize SGX-enabled applications. Hence, we decided to leverage both PRSA and MIR solutions within our *SGXTuner* that further prove the effectiveness of the proposed approach.

6.3 The SPISA Algorithm

The nature of the optimization problem covered by *SGXTuner* led us to design a dedicated parallel SA algorithm, which maintain high parallelism efficiency and at the same time do not upset the state generation, i.e., preserve the convergence properties.

The *Simultaneous Periodically Interacting Simulated Annealing* (SPISA) algorithm is positioned in the middle between *serial-like* and *altered-generation* procedures. SPISA aims at keeping unaltered the SA properties by making small variations to the original flow. It evaluates in parallel, and without any communication, a *neighborhood* space $\mathcal{N}$ composed of *one-exchange* states parameters. That is, the set of all states (or parameters) generated from the current selected best one s through the modification of one single parameter value at a time. When the $\mathcal{N}$ set runs out, SPISA synchronizes results obtained from workers by comparing the current best *master* s with the s_w having the best energy among all workers.

The three boxes reported below describe the algorithm carried out by both *master* and *workers*. The function receives in input the space state S, the initial state s_0, the temperature T_0, the cooling strategy Ψ, the maximum number of steps to execute k_{max}, the maximum value of subsequent rejected improvements of the best energy rej_{max}, and the number of *worker* instances n_w to launch.

SPISA Algorithm - Master

1: Evaluate E of initial s and define *shared* temperature T
2: Iterate until convergence or the maximum number of execution steps is reached:
  a) Initialize a *pool* of neighbourhood $\mathcal{N}(s)$
  b) Spawn n_w workers
  c) Send T and $\mathcal{N}(s)$ to workers
  d) Receive n_w workers' best states s_w^i
  e) Verify *Boltzman* criterion between s and best s_w
  f) Update T

After the initialization of s, T, and $E(s)$, a *shared pool* of $\mathcal{N}(s)$ states is created (line 5). Each *worker* — as soon as it gets free — extracts from the pool a random state to perform its evaluation. In this way, the processing rate is always kept high and the *worker* idle time is significantly low.

SPISA Algorithm - Worker

1: Extract random state s_w from $\mathcal{N}(s)$ and evaluate E_w
2: Iterate until $\mathcal{N}(s) \neq \emptyset$
  a) Extract random s'_w from $\mathcal{N}(s)$ and evaluate E'_w
  b) Verify *Boltzman* criterion between s_w and s'_w
  c) Update T
3: Return s_w to the Master

SPISA Algorithm Procedure

1: **function** SPISA($S, s_0, T_0, \Psi, k_{max}, rej_{max}, n_w$)
2: $s \leftarrow s_0; T \leftarrow T_0$
3: $E(s) \leftarrow Eval(s)$
4: **while** $k \leq k_{max} \wedge rej \leq rej_{max}$ **do**
5: $init\ Pool(\mathcal{N}(s))$
6: $SpawnWorkers(n_w)$
7: ---
8: $s_w \leftarrow Pool(\mathcal{N}(s)).pop()$;
9: $E_w(s_w) = Eval(s_w)$
10: **while** $|Pool(\mathcal{N}(s))| > 0$ **do**
11: $s'_w = Pool(\mathcal{N}(s)).pop()$
12: $E'_w(s'_w) = Eval(s'_w)$
13: $\Delta E_w = E_w(s_w) - E'_w(s'_w)$
14: **if** $\Delta E_w \leq 0 \vee R(0,1) \leq e^{-\Delta E_w/T}$ **then**
15: $(s_w, E(s_w)) \leftarrow (s'_w, E'_w(s'_w))$
16: **if** $\Delta E_w \leq 0$ **then**
17: $rej \leftarrow 0$
18: **end if**
19: **else**
20: $rej \leftarrow rej + 1$
21: **end if**
22: $T \leftarrow \Psi(T, k)$
23: $k \leftarrow k + 1$
24: **end while**
25: ---
26: $s' \leftarrow s_w : E(s_w) = max(\widehat{E_w})$
27: $E'(s') \leftarrow max(\widehat{E_w})$
28: $\Delta E = E(s) - E'(s')$
29: **if** $\Delta E \leq 0 \vee R(0,1) \leq e^{-\Delta E/T}$ **then**
30: $(s, E(s)) \leftarrow (s', E'(s'))$
31: **end if**
32: **end while**
33: **return** s
34: **end function**

Then, n_w *workers* are spawned. Line 8 to 24 are executed in parallel by each of them. State s'_w is evaluated and then used with s_w to verify the *Boltzmann criterion*. In case $\Delta E_w > 0$ the number of subsequent rejected states rej is increased. Otherwise, if $\Delta E_w \leq 0$, rej is reset to 0. In a nutshell, if better energies are found, rej is reset. If not, it is incremented.

Afterwards, in line 22-23, the temperature and step counter are updated. It is important to notice that rej, T, and k are shared variables between workers. This allows to maintain the parallel flow as much as possible near to the sequential one.

When the *pool* becomes empty, i.e., the neighborhood exploration is terminated, the *master* takes control again. At this point, the s_w having maximum energy among n_w workers is chosen to be compared with best state s. Hence, an additional verification of *Boltzmann criterion* is enforced between s and s_w.

SPISA keeps iterating until the number of maximum steps k_{max} or the maximum subsequent rejected states rej_{max} are not reached. The latter case would mean that SPISA converged to an optimum before exhausting the number of allowed steps.

The critical aspect of sequential SA, which makes difficult its parallelization, is the exploration of the neighborhood space. At every iteration, a new state contains specific modifications — defined by the chosen *neighborhood structure* — to the previous one. Therefore, in parallel algorithms, it is important to guarantee that neighborhoods explored by a i-th worker are also neighborhoods for the other workers, and that they can be independently analysed. The rationale behind SPISA is to build a limited neighborhood space, in which all the elements are neighbor between them, and leaving to workers the exploration of states belonging to. At the end of their task, the *master* compares solutions temporarily selected as optimal by *workers*.

What is important in SPISA is the definition of neighborhood and the independence analysis of its elements. Let P_i be the i-th configuration parameter having a varying number of values $\mathcal{V}$, and N_P the number of possible parameters. A state s, at any point of the tuning process, is composed of a combination of N_P parameter values. The space state S, instead, is the vector containing all the possible combinations of the N_P parameter values, having size $|S| = \prod_{i=1}^{N_P} |P_i|$. We say that s is neighbor of s' if s differs for at most two parameters from s'. Thus, we create from s a space $\mathcal{N}(s)$ built with *one-exchange* parameters, which is therefore a subset of the whole s neighborhood space and has cardinality: $|\mathcal{N}| = \sum_{i=1}^{N_P} (|P_i| - 1)$. It can be proven that $\mathcal{N}(s)$ contains states, which at most differ for two parameters. Therefore, the $|\mathcal{N}|$ states belonging to $\mathcal{N}(s)$ are all neighborhood between them.

7 Evaluation

A massive evaluation activity has been conducted on three applications widely used in commercial solutions: we found optimal configurations of *sgx-musl* using *Memcached*, *Redis*, and *Apache*. The *objective function* (Ω) is the maximization of throughput, and the *Energy* (E) refers to the number of operations per unit time ($Kops/s$). We compared results coming from the sequential variant (SEQSA) and the three parallel SA algorithms described in section 6, i.e., SPISA, MIR, and PRSA. In the rest of this section, we first introduce the experimental testbed, then we present the baseline used as a reference for results comparison, and finally we show and discuss the outcomes of the experimental campaign in terms of performance gain, parameters weight, and convergence properties.

7.1 Testbed

The evaluation was realized with three *Target* applications built with *sgx-musl*, i.e., *Memcached*, *Redis*, and *Apache*. These represent two different typologies of distributed client-server applications. The first two are in-memory *key/value* storage systems with a multi-thread and single-thread architecture, respectively. The third one is a multi-threaded *HTTP* web server. Besides the *Target*, an additional choice relates to the *Benchmarking* tools used as *Workload* generator to be adopted for conducting the

experiments. In this regard, we solicit *Memcached*, *Redis*, and *Apache* with three widely-accepted benchmarking tools: *Memaslap*[1], *Memtier* [2], and *WRK*[3].
Figure 3 shows the experimental set up. Each *Target* executes on a separate physical SGX Node. This choice has been driven by the need of having undisturbed performance measurements. In fact, running n_w parallel procedures on a single multi-core machine means executing n_w instances of Memchached/Redis/Apache. This entails a significant overhead causing altered measurements. Therefore, to reduce interferences we decided to use n_w different servers.

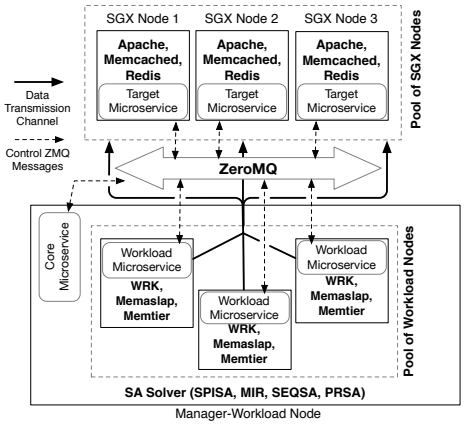


Fig. 3: Testbed used for the evaluation activity

The n_w correspondent *benchmarking* tools, instead, run on a single Manager-Workload Machine. In this case, the mutual disturbance they generate between each other is negligible as the server on which they run is highly performing.
The different microservices are deployed within *Docker* containers and are clustered through *Docker Swarm*.

Three identical machines — having the Intel SGX extension — have been employed. These are composed of an Intel Xeon E3-1270 v5 CPU with 4 cores at 3.6 GHz, 8 hyper-threads (2 per core), and 8 MB cache. The server has 64 GB of memory and runs Ubuntu 14.04.4 LTS with Linux kernel version 4.2. The Manager-Workload Node, instead, has two 14-core Intel Xeon E5-2683 v3 CPUs at 2 GHz with 112 GB of RAM and Ubuntu 15.10. Each machine has a 10 Gb Ethernet NIC connected to a dedicated switch.

7.2 Baseline Configuration

The baseline configuration of *sgx-musl* — needed as a reference for measuring the performance improvement of *Memcached*, *Redis*, and *Apache* — was defined using our detailed knowledge of internal mechanisms of *sgx-musl* and on the results of preliminary micro-benchmarks. More precisely, for each application we performed a set of experiments using the two execution modes of SGX, i.e., simulation and normal modes. In simulation mode, the application uses untrusted and unprotected memory, as opposed to EPC memory, which is confidentiality and integrity protected by the hardware. Experiments performed using simulation mode allowed us to eliminate the overhead caused by the memory protection mechanism and assess performance of the *sgx-musl syscall* interface in isolation. We manually performed experiments and measured both throughput and latency to define satisfactory values. We slightly modified the parameters and observed how the new values affect the performance. This procedure was repeated several times until a reasonably good configuration was obtained.
Both *Memcached* and *Apache* are multi-threaded applications, thus, according to the considerations made in section 4, we initially chose to the number of `ETHREADS` to exactly half of the number of Intel's CPU hyper-cores ($= 4$). The number of `STHREADS` was chosen to be multiple of application-level threads (around 30). *Redis*, instead, is single-threaded. Hence, in this case `ETHREADS` is set to 1, while we kept `STHREADS` to the minimum as we noticed better performance for such low values. We also observed a good trend when `ESPINS` and `ESLEEP` are low for single-threaded and high for multi-threaded software. This seems reasonable as fewer threads should sleep less time. The selected baseline values are presented below.

Baseline Parameters (Single/Multi)

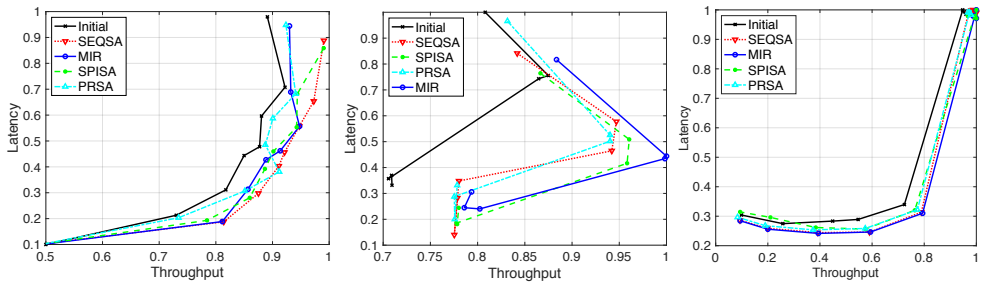
`ETHREADS=1/4`	`ESPINS=3,5k/4,8k`	`ESLEEP=15k/18k`
`STHREADS=20/35`	`SSPINS=80/80`	`SSLEEP=3,4k/3,4k`

7.3 Performance Gain

Overall, both *Memcached* and *Apache* built with *sgx-musl* showed a substantial performance gain, when optimal configurations produced by *SGXTuner* were used. *Redis*, instead, reported a minor gain (see Figures 4). For each parameter setting that the different stochastic algorithms found, we report the overall throughput and the per-request latency for increasing workloads. In fact, we want to demonstrate that the benefit of *SGXTuner* optimization to the *sgx-musl* overhead is significant for both metrics, regardless of the selected *objective function* (i.e., minimization/maximization of latency/throughput). *Memcached* and *Redis* were solicited for $20s$ with $[32, 512]$ concurrent *key/value* requests characterized by 16 parallel threads. *Apache*, instead, received for $20s$ $[5, 50]$ HTTP concurrent GET requests characterized by 8 parallel threads. Both applications were evaluated 10 times and the results averaged through a 25% trimmed mean, which produces robust estimates. The reported graphs are normalized to the *best* — meaning highest and lowest, respectively — observed values of throughput and latency.

Regarding *Memcached* (Figure 4a), best results of throughput are reached with SEQSA and SPISA parameters using 512 concurrent requests, which also represent the saturation point reached by *Memcached* with such configurations. In fact, among the different top-right marks reported in the graph for 512 requests, SEQSA and SPISA are the rightmost and lowest ones. In such a case, SEQSA provided a rate of $331k.ops/s$ and a latency of $1.52ms$. The throughput-latency performance gain of SPISA is of 11.05-12.13% with respect to the baseline configuration defined by a domain expert, while it is of 45.41-39.36% if the reference taken as baseline is a worse configuration, which could be selected randomly by a non-expert user. The graph

1. http://docs.libmemcached.org/bin/memaslap.html
2. https://github.com/RedisLabs/memtier_benchmark
3. https://github.com/wg/wrk



(a) *Memcached* (b) *Apache* (c) *Redis*

Fig. 4: Throughput vs Latency using *SGXTuner* optimal parameters

also demonstrates that even MIR and PRSA parameters are able to improve the performance of the SGX-enabled *Memcached*, even though the gain of throughput-latency is slighter, e.g., 5.21%-4.46% using PRSA with respect to the good *Initial* setting. Furthermore, unlike SEQSA and SPISA, the saturation point with these two configurations is reached at 320 concurrent requests, after which the throughput decreases and therefore the trend line moves towards the left direction.

For what concerns *Apache* (Figure 4b) a different behavior was observed. First of all, we can notice larger differences in terms of latency. In fact, the graph shows that markers of the *Inital* baseline configuration are always quite higher than markers of the configurations found by *SGXTuner*. For all of them, the saturation point is the same, i.e., 40 HTTP concurrent GET requests sent to the web server. After that value, the throughput tends to decrease as demonstrated in the graph from all lines going towards the left direction. The best outcomes were obtained when configurations found by *SGXTuner* with MIR and SPISA are used. In the case of MIR, the gain in terms of throughput-latency is of $14.31\% - 37.14\%$ with respect to the good *Initial* configuration, while it is of $38.77\% - 51.25\%$ compared to a random configuration of *sgx-musl*. It must be noticed that the improvement in terms of throughput is lower than the one obtained for latencies but it is still significant. SPISA instead yielded performance results, which represent a gain of 9.79%-23.28% with respect to good and random baselines 27.47-35.91%.

Redis, instead, reported a slighter performance enhancement, which in the best case (i.e., MIR) ranges from $\approx$ 7%-5% to $\approx$ 13%-10%. As shown in Figure 4c, differences in graphs' trends are much less evident than in previous cases. This leads us to assume that applications with single-threaded architectures have narrower margins of improvement.

Outcomes obtained during this experimental evaluation demonstrate that our approach is always useful to reduce the overhead introduced on *Target* applications by *system-level* SGX solutions. Regardless of the chosen stochastic algorithm, there is always an improvement of both throughput and latency. We noticed that the specific choice on the *objective function* does not influence the improvement of the particular metric. Rather, in some cases (e.g., *Apache*) even if the maximization of throughput was selected in *SGXTuner*, better latencies were observed.

7.4 Convergence of *SGXTuner*

In this subsection, we analyze the way *SGXTuner* converges to the final optimal solution as time grows. Figures 5 and 6 report the smoothed trend of *Target*'s throughput (or *Energy*) for increasing exploration time, when parameters temporarily selected as optimal are used. For simplicity, we only show the trend of *Memcached* and *Apache*.

In general, we can immediately notice that the time required to reach the optimal *sgx-musl* parameters was very short compared to the time needed to test the whole subspace of configurations presented in section 4. The sequential SA (SEQSA) took more time ($\approx 28Hrs$ for Memcached and $\approx 22Hrs$ for Apache) than the parallel heuristics. The speed-up of using three SGX Nodes was mainly linear (2.97 to the best) and super-linear in only few cases (e.g., 3.15 during the SPISA run for Memcached). A separate discussion is needed for MIR. This, in fact, has a time of completion similar to SEQSA. This is not surprising since MIR spawns multiple independent sequential SA jobs, and the tuning process time depends on the slowest worker. Therefore, in terms of absolute time MIR spends an amount of time near to the sequential one. However, the usage of n_w parallel *workers* has the potential of finding better performing solutions with the same number of tuning steps. This is due to the higher heterogeneity of the state space searched by the parallel jobs. The PRSA algorithm, instead, was more time-consuming ($\approx 12Hrs$) than SPISA. This was expected as PRSA — unlike SPISA and MIR — alters the native SA algorithm, which entails different convergence properties.

As expected, *SGXTuner* starts selecting bad configurations with low throughput and shrink to better solutions after $\approx 1/3$ of execution time. In fact, SA has a significant probability of accepting a downhill move during the early steps. It is worth to observe that, in a few cases (e.g., Fig. 5a), optimal parameters yielded at the end of tuning activities have a throughput, which is lower than others obtained during the state space exploration. This in principle should not happen as *SGXTuner* is meant to find the global optimum. However, it is not surprising since we set the SA settings (i.e., initial temperature T_0, the cooling strategy, and the termination criterion) with the goal of finding the best trade-off between the quality of the solution and the convergence time. It is the consequence of the stochastic nature of SA, where randomness is involved. Ideally, in larger time windows, with high temperatures, an extremely

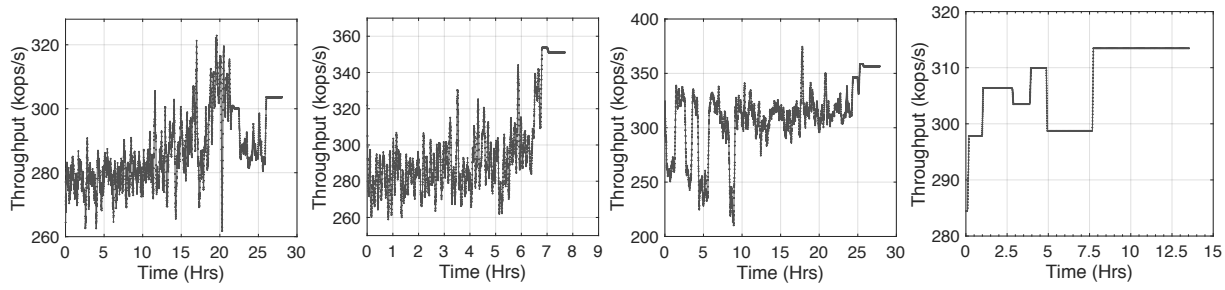


Fig. 5: *Memcached* - Throughput of configurations selected as optimal - SEQSA (5a), SPISA (5b), MIR (5c), and PRSA (5d).

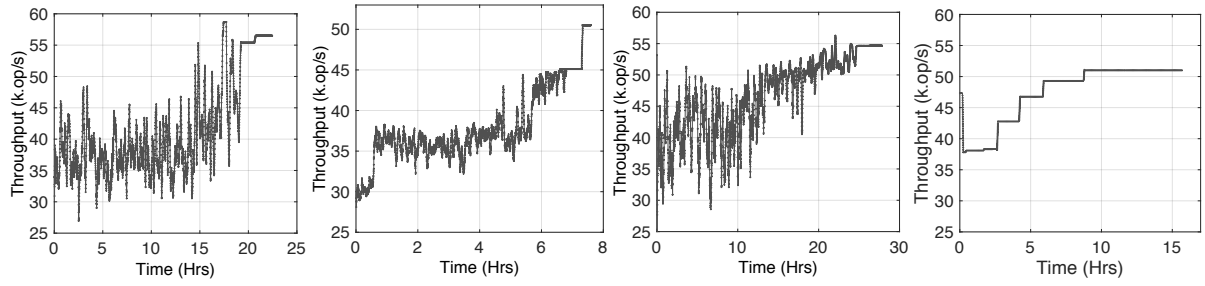


Fig. 6: *Apache* - Throughput of configurations selected as optimal - SEQSA (6a), SPISA (6b), MIR (6c), and PRSA (6d).

Parameters	SEQSA			SPISA			PRSA			MIR		
	M	R	A	M	R	A	M	R	A	M	R	A
STHREADS	60	40	45	50	70	50	50	25	50	55	65	50
ETHREADS	4	1	3	3	1	3	4	1	3	3	1	3
SSLEEP	4k	4k	3,3k	3,7k	4k	3,9k	3,5k	3,8k	3,2k	3,3k	4k	3,7k
ESLEEP	16k	18k	17k	16k	18k	16k	17k	20k	16k	16k	19k	18k
SSPINS	90	110	140	120	100	130	100	100	130	140	110	120
ESPINS	4,6k	5,6	5,2K	4,6k	5,6	5,1k	4,5k	5,6	4,9k	4,3k	5,4	5,2k

TABLE 2: Parameters of *Memcached* (M), *Redis* (R), *Apache* (A)

slow temperature cooling, and a favorable termination criterion, this behavior should not occur. We believe that sacrificing the convergence properties to obtain lower execution time is acceptable for the type of optimization problems managed by *SGXTuner*.

7.5 Parameters Analysis and Hints

Table 2 reports the optimal *sgx-musl* parameters, which were obtained for the three applications tuned with the different stochastic algorithms. Results confirm that *SGXTuner* has global exploration abilities. In fact, for each specific *Target*, the combination of parameters' values are very similar between each other. There are only small-scale variations, which we think are caused by the intrinsic variance in *Target* software performance. Slightly different configurations have a limited impact on applications' throughput. We can notice that the optimal configuration of *sgx-musl* depends on the type *Target* application. In fact, *SGXTuner* produced different solutions for *Memcached*, *Redis*, and *Apache*.

Regarding *Memcached* and *Apache*, we can notice that `SSLEEP` and `SSPINS` are in a larger range of values. This could lead to suppose that these parameters should not affect the overall performance. However, we can observe that high values of `SSPINS` entail low values of `SSLEEP`, and vice-versa. Thus, their combined configuration should be of significance. Contrariwise, the `ETHREADS` and `STHREADS` parameters reported very similar values for the two multi-threaded applications. Regarding *Redis*, a remarkable observation is that our assumptions of using low values for `STHREADS` was wrong as all the optimal parameters produced by the different algorithms are much higher (≈ 70). While, for all the SA algorithms, `ETHREADS` is equal to 1, as expected. Unexpectedly, we obtained higher values of `ESPINS` and `ESLEEP` when used in single-threaded software.

To better understand the impact of each parameter on the overall performance, we performed a feature selection for regression using the dataset of optimal solutions obtained from the search space exploration made by *SGXTuner*. Parameters' weights are learned by a diagonal adaptation of Neighborhood Component Analysis (NCA) with regularization. Figure 7 shows the classification for the three *Target* applications. Parameters used in Single- and Multi- threaded software have different weights. Results shows that `SSLEEP` and `SSPINS` have lower weight when used with multi-threaded applications. The bar graph also shows that `ESLEEP` and `ESPINS` have significant impact on *Memcached* and *Apache* performance while they have less weight in the case of *Redis*. Moreover, `ETHREADS` has the highest impact when used with the single-threaded *Redis*. In fact, using values $\neq 1$ could drastically reduce the throughput. Finally, `STHREADS` resulted in the most important parameter for *Apache*'s performance.

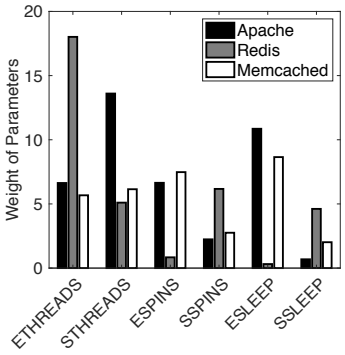


Fig. 7: Weights of parameters

8 Related Work

To the best of our knowledge, there are no research works proposing a generic approach for performance improvement of *system-level* solutions enabling transparent SGX support. Stochastic optimization is adopted in a number of engineering fields for parameters tuning, but not for the type of problems covered in this work. Three investigations are close to our paper. On a more generic level, Zhong et *al.* [26] proposed a tool called *AcovSA*, which finds optimized compiler options with SA to improve the performance of applications compiled with *GCC*. Authors tuned $\approx$ 60 *GCC* options belonging to the four levels of OOS, i.e., O_1, O_2, O_3. Experimental results showed the efficiency and effectiveness of the proposed tuning tool. In this work, compiler-related parameters represent only a subset of those that could be taken into account by *SGXTuner*. Zhao et *al.* [27] made a basic evaluation of SGX runtime performance by identifying the sources of overhead when different execution modes and parameters are used. Authors compared the performance of the most important *libc* functions when executed inside and outside the enclave. *SGXTuner* — unlike this paper — is a solution that supports the performance enhancement. Finally, even Weichbrodt et *al.* [28] identified critical overhead factors of SGX via a tool called *sgx-perf*, whose main goal is to obtain a fined-grained profiling of performance events in enclaves. Authors propose also a way for optimizing the enclave performance which produced a low increase ($\approx 2\%$). *SGXTuner* — unlike *sgx-perf* — provides a targeted solution for *system-level* SGX approaches, the most used ones by research and industrial communities. Moreover, *SGXTuner* reported higher performance improvement.

9 Concluding Remarks

This paper provides a valuable contribution to Intel SGX ecosystems. It demonstrates the effectiveness of a general approach to optimize the performance of applications secured with the current state-of-the-art of *system-level* solutions for transparent SGX support, whose key feature is the use of a SGX-extended *libc* library. We proposed *SGXTuner*, a convenient solution for reducing the configuration time of SGX-extended *libc* libraries, which are tunable with many parameters essential for the overall performance. *SGXTuner* does so by availing itself of stochastic optimization. A thorough experimental campaign was conducted on a particular implementation of *SGXTuner* using sequential and parallel variants of *Simulated Annealing*.

The tool found settings of *sgx-musl*, which enabled high-performance of the SGX-extended *libc* library. This directly impacts a large fraction of applications using SGX transparent support, since *libc* is at the base of the widely-accepted SCONE and SGX-LKL system-level approaches. Three commercial web applications — i.e., *Memcached*, *Redis*, and *Apache* — were used for the experimental evaluation. Among different SA algorithms, the parallel SPISA scheme reported the best trade off between rate of convergence and performance results.

The outcomes of our experiments corroborate the contribution of this work. It was proved that *SGXTuner* is able to significantly improve the performance of applications secured with *system-level* SGX solutions. Multi-threaded software such as *Memcached* and *Apache* reported substantial gain, while single-threaded ones such as *Redis* showed a slighter improvement. It was demonstrated that *SGXTuner* can support the identification of parameter combinations that would not have been visible with manual tuning. We showed that some of the assumptions we made during initial tests turned out to be wrong. We analyzed parameters' weight for each type of application. We discovered that some combinations of parameters are important for single-threaded software, while others have more impact when used for multi-threaded applications. The statistical analysis conducted on the optimal parameters dataset obtained by *SGXTuner* showed that the performance of the single-threaded *Redis* is more dependent on the values of `ETHREADS`, `STHREADS`, and `SSPINS`. While, for the multi-threaded *Memcached* and *Apache*, `STHREADS`, `ESLEEP` and `ESPINS` have higher impact.

In practical terms, the findings of this paper come handy to the large community of SGX users/developers. They can tune any of the different available *system-level* SGX solutions to harden their application of interest while also ensuring high levels of performance. The community could adopt the proposed method, or the *SGXTuner* tool itself, which moreover is open source on GitHub[4]. They could leverage findings from the parameters analysis to set much better initial configurations or use them as definitive ones, especially if the application is single-threaded thus it has fewer margins for improvement. We explicitly emphasize that even larger improvements can be achieved by simply extending our approach to *application-related* or *compiler-related* parameters (discussed in Section 3). This can be done by including these parameters in the tuning process. The potential improvement is particularly significant in the case of single-threaded applications, where parameters external to the enclave have been demonstrated to have a direct impact on performance. In our experimental activities we considered those related to the specific SGX-extended *libc* library (i.e., *sgx-musl*), and the gain of throughput-latency has been significant: 11-12%, 14-37%, 7%-5% for *Memcached*, *Apache*, and *Redis*, respectively.

4. https://github.com/dzobbe/sgxtuner

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 6450311 (SERECA), and from the "ICShield" project sponsored by the Italian Ministry of Economic Development.

REFERENCES

[1] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, C. Fetzer, Scone: Secure linux containers with intel sgx, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, USENIX, 2016, pp. 689–703.
URL http://dl.acm.org/citation.cfm?id=3026877.3026930

[2] C. che Tsai, D. E. Porter, M. Vij, Graphene-sgx: A practical library OS for unmodified applications on SGX, in: 2017 USENIX Annual Technical Conference (USENIX ATC 17), USENIX Association, Santa Clara, CA, 2017, pp. 645–658.
URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[3] SGX-LKL Library OS for Running Java Applications in Intel SGX Enclaves, https://github.com/lsds/sgx-lkl, 27/05/2018.

[4] A. Baumann, M. Peinado, G. Hunt, Shielding applications from an untrusted cloud with haven, ACM Trans. Comput. Syst. 33 (3) (2015) 8:1–8:26. doi:10.1145/2799647.

[5] M. Orenbach, P. Lifshits, M. Minkin, M. Silberstein, Eleos: Exitless os services for sgx enclaves, in: Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17, ACM, New York, NY, USA, 2017, pp. 238–253. doi:10.1145/3064176.3064219.

[6] T. Hunt, Z. Zhu, Y. Xu, S. Peter, E. Witchel, Ryoan: A distributed sandbox for untrusted computation on secret data, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA, 2016, pp. 533–549.

[7] S. T. S. Shinde, D. L. Tien, P. Saxena, Panoply: Low-tcb linux applications with sgx enclaves, in: Proceedings of the Ninth European Conference on Computer Systems, NDSS, 2017.

[8] Microsoft Azure Confidential Computing, https://azure.microsoft.com/it-it/solutions/confidential-compute/, 22/01/2019.

[9] IBM Cloud Data Shield, https://www.ibm.com/cloud/data-shield, 20/01/2019.

[10] SGX-protected Alibaba Cloud Offering, https://www.alibabacloud.com/blog/fortanix-provides-intel%C2%AE-sgx-protected-kms-with-alibaba-cloud_594075, 22/01/2019.

[11] Musl Library, https://www.musl-libc.org/, 27/05/2018.

[12] GNU C Library project, https://www.gnu.org/software/libc/, 27/05/2018.

[13] Intel - Firmware Updates and Initial Performance Data for Data Center Systems, https://newsroom.intel.com/news/firmware-updates-and-initial-performance-data-for-data-center-systems/, 20/02/2019.

[14] PostgreSQL: Heads Up: Fix for intel hardware bug will lead to performance regressions, https://www.postgresql.org/message-id/20180102222354.qikjmf7dvnjgbkxe@alap3.anarazel.de, 20/02/2019.

[15] P. J. M. van Laarhoven, E. H. L. Aarts, Simulated annealing, Springer Netherlands, Dordrecht, 1987, pp. 7–15. doi:10.1007/978-94-015-7744-1_2.

[16] Y. Nourani, B. Andresen, A comparison of simulated annealing cooling strategies, Journal of Physics A: Mathematical and General 31 (41) (1998) 8373.
URL http://stacks.iop.org/0305-4470/31/i=41/a=011

[17] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, U. R. Savagaonkar, 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, ACM, New York, NY, USA, 2013, pp. 10:1–10:1. doi:10.1145/2487726.2488368.

[18] Fortanix Runtime Encryption, https://fortanix.com/products/runtime-encryption/, 22/01/2019.

[19] L. Coppolino, S. D'Antonio, G. Mazzeo, L. Romano, A comparative analysis of emerging approaches for securing java software with intel sgx, Future Generation Computer Systemsdoi:https://doi.org/10.1016/j.future.2019.03.018.
URL http://www.sciencedirect.com/science/article/pii/S0167739X18315942

[20] O. Purdila, L. A. Grijincu, N. Tapus, Lkl: The linux kernel library, in: 9th RoEduNet IEEE International Conference, 2010, pp. 328–333.

[21] S. W. Mahfoud, D. E. Goldberg, Parallel recombinative simulated annealing: A genetic algorithm, Parallel Computing 21 (1) (1995) 1 – 28. doi:http://dx.doi.org/10.1016/0167-8191(94)00071-H.

[22] F.-H. A. Lee, Parallel simulated annealing on a message-passing multi-computer, Ph.D. thesis, Logan, UT, USA, uMI Order No. GAX96-20327 (1995).

[23] J. Olenšek, T. Tuma, J. Puhan, A. Brmen, A new asynchronous parallel global optimization method based on simulated annealing and differential evolution, Appl. Soft Comput. 11 (1) (2011) 1481–1489. doi:10.1016/j.asoc.2010.04.019.

[24] D. R. Greening, Parallel simulated annealing techniques, Phys. D 42 (1-3). doi:10.1016/0167-2789(90)90084-3.

[25] K. F. Man, K. S. Tang, S. Kwong, Genetic algorithms: concepts and applications [in engineering design], IEEE Transactions on Industrial Electronics 43 (5) (1996) 519–534. doi:10.1109/41.538609.

[26] S. Zhong, Y. Shen, F. Hao, Tuning compiler optimization options via simulated annealing, in: 2009 Second International Conference on Future Information Technology and Management Engineering, 2009, pp. 305–308. doi:10.1109/FITME.2009.81.

[27] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, C. Xing, On the performance of intel sgx, in: 2016 13th Web Information Systems and Applications Conference (WISA), 2016, pp. 184–187. doi:10.1109/WISA.2016.45.

[28] N. Weichbrodt, P.-L. Aublin, R. Kapitza, Sgx-perf: A performance analysis tool for intel sgx enclaves, in: Proceedings of the 19th International Middleware Conference, Middleware '18, ACM, New York, NY, USA, 2018, pp. 201–213. doi:10.1145/3274808.3274824.

Giovanni Mazzeo , PhD, is Assistant Professor at the Department of Engineering of the University of Naples "Parthenope". His research activity mainly focuses on security and dependability of computer systems. He is actively involved in European projects on IT security.

Sergei Arnautov , PhD, is a research assistant at the Systems Engineering Chair in the Computer Science Department at the Dresden University of Technology. His research interests are dependability, distributed systems, fault tolerance, and system security. He is involved in European research as well as industry projects on IT security.

Christof Fetzer , PhD, is Full Professor and head of the Systems Engineering Chair in the Computer Science Department at the Dresden University of Technology. He is the chair of the Distributed Systems Engineering International Masters Program at the Computer Science Department.

Luigi Romano , PhD, is a Full Professor at the University of Naples "Parthenope". His research field is system security and dependability, with focus on Critical Infrastructure Protection. He was one of the experts appointed by ENISA for aligning research to the NIS directive. He works extensively as a consultant for industry leaders in the field of critical computer systems.