

A Comprehensive Benchmark Suite for Intel SGX

Sandeep Kumar
School of Information Technology
IIT Delhi
New Delhi, India
sandeep.kumar@cse.iitd.ac.in

Abhisek Panda
Department of Computer Science and
Engineering, IIT Delhi
New Delhi, India
abhisek.panda@cse.iitd.ac.in

Smruti R. Sarangi
Department of Computer Science and
Engineering, IIT Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

ABSTRACT

Trusted execution environments (TEEs) such as Intel SGX facilitate the secure execution of an application on untrusted machines. Sadly, such environments suffer from serious limitations and performance overheads in terms of writing back data to the main memory, their interaction with the OS, and the ability to issue I/O instructions. There is thus a plethora of work that focuses on improving the performance of such environments – this necessitates the need for a standard, widely accepted benchmark suite (something similar to SPEC and PARSEC). To the best of our knowledge, such a suite does not exist.

Our suite, *SGXGauge*, contains a diverse set of workloads such as blockchain codes, secure machine learning algorithms, lightweight web servers, secure key-value stores, etc. We thoroughly characterize the behavior of the benchmark suite on a native platform and on a platform that uses a library OS-based shim layer (GrapheneSGX). We observe that the most important metrics of interest are performance counters related to paging, memory, and TLB accesses. There is an abrupt change in performance when the memory footprint starts to exceed the size of the EPC size in Intel SGX, and the library OS does not add a significant overhead ($\approx \pm 10\%$).

1 INTRODUCTION

Intel Secure Guard eXtension or Intel SGX [5, 29] has gained popularity in recent years as a way to securely execute an application on a remote, untrusted machine. The security of the application and data within SGX, i.e., confidentiality, integrity, and freshness are guaranteed by the hardware. The code and data within Intel SGX is even out of the reach of privileged system software such as the operating system and hypervisor. Recently, Microsoft Azure adopted Intel SGX to provide secure computing in their data centers [2, 67].

However, this protection comes at a cost. Intel SGX, to ensure security guarantees, puts certain restrictions on the applications running within it, such as no system calls, as the operating system is not a part of the trusted framework of SGX [29]. Therefore, few additional and expensive steps are required to enable system call support, which incurs additional performance overheads [36, 80]. Furthermore, Intel SGX reserves a portion of the main memory for its operations, which is managed by the hardware. However, this reserved memory is limited in size, and any application allocating more than the reserved memory, incurs a significant amount of performance overhead [29, 51].

Researchers have focused on alleviating this problem by proposing different mechanisms and workarounds to reduce the overheads [51, 62, 68, 72, 73, 80]. To show the benefits of their methods, researchers have resorted to manual porting of applications to Intel

SGX [32, 36]. However, porting an application requires significant expertise and development effort [36]. Also, the decision of which application to port is generally motivated by the ease of porting, and not necessarily by the gains accrued by doing so. Hence, there is no accepted, standard method for benchmarking SGX-based systems primarily due to the ad hoc nature of workload creation.

The big picture is as follows. The workloads used to evaluate the efficiency of different methods for improving Intel SGX vary across different proposals. Hence, it is not possible to compare the performance gains in one work with those of another in any meaningful manner. Therefore, there is a need for a standard benchmark suite for Intel SGX, much like traditional benchmarks suites such as SPEC [49] and PARSEC [16].

A benchmark suite needs to thoroughly evaluate all the critical components of Intel SGX, and enable performance comparison by setting a common denominator across different works. Primarily, there are three sources of performance overheads in Intel SGX: **encryption/decryption of the data in the reserved secure memory, the cost for accessing operating system services, and the additional time for swapping in data when an application has allocated more memory than the reserved memory** [51, 80]. Prior works [32, 36, 53] in this field propose different benchmark suites [19, 56] for evaluating Intel SGX. However, they only focus on the first two costs, ignoring the last one, which accounts for the maximum performance overhead [51].

We present *SGXGauge* – a comprehensive benchmark suite for Intel SGX. *SGXGauge* contains 10 real-world and synthetic benchmarks from different domains that thoroughly evaluate all the critical components of Intel SGX. We use *SGXGauge* to evaluate Intel SGX in two different modes: ① *native* mode where we port the benchmarks to Intel SGX, and ② *shim* mode where we execute benchmarks in an environment where a thin system software layer intercepts the system calls and intercedes with the OS on behalf of the application [24, 68, 70]. Such shim layers are also known as library operating systems; they are gaining popularity because they significantly reduce the development time required to run an application on SGX as compared to porting the same application to SGX [36]. Our precise list of contributions are as follows.

- (1) We present *SGXGauge*, a benchmark suite for Intel SGX that thoroughly evaluates all of its components.
- (2) We stress test the impact of EPC on the performance of applications – a crucial component that is missing from prior work.
- (3) We thoroughly evaluate the performance overhead incurred while executing an application with a library operating system.

The rest of the paper is organized as follows: we discuss the required background for the paper in Section 2. In Section 3 we discuss related work and the motivation for the paper. This is followed by a detailed overview of our benchmark suite, SGXGauge, in Section 4. We discuss the evaluation of the benchmarks in Section 5. We finally conclude in Section 6.

2 BACKGROUND

In this section, we discuss the necessary background for the paper.

2.1 Intel SGX

Intel Secure Guard eXtension or SGX ensures the secure execution of an application either on a local or remote machine. It guarantees confidentiality, integrity, and freshness of the code and data running within it. Even privileged system software such as the operating system and hypervisor cannot affect its execution.

SGX reserves a part of the system memory for its use at boot time. This reserved memory is known as the *Processor Reserved Memory* or PRM [29]. Our system supports 128 MB of PRM, and the rest of the discussion in the paper is based on this setting. The PRM is split into two regions that are used to store ❶ SGX metadata and ❷ data/code of user applications, respectively. The latter is called the *Enclave Page Cache* or EPC. The size of the EPC is 92 MB, although SGX supports applications that require more memory (details in Section 2.2). For every process, SGX creates a trusted execution environment an *enclave* [29].

The operating system cannot access the data within an enclave. However, an enclave still requires the operating system’s support for setting it up, scheduling, context switching, page management, and cleanup. To enable this, the memory management of the enclave is done by the operating system. Just before launching an enclave, the hardware checks the loaded binary for tampering by securely calculating its signature (hash) and matching it with the signature provided by the enclave’s author.

2.2 Enclave Page Cache

The EPC is used to allocate memory for all the applications executing within SGX. The data in the EPC is always in an encrypted form to prevent any snooping from privileged system software such as the operating system. The data is decrypted when brought in to the LLC (last level cache) upon a CPU request. Intel SGX uses a dedicated hardware called the *Memory Management Engine* or MEE for encrypting and decrypting the data.

Sadly, the size of the EPC is one of the major limitations of SGX [51]. A typical modern application generally has a working set that is more than 92 MB [35, 60]. In such cases, SGX transparently evicts pages from the EPC to the untrusted memory, albeit in an encrypted form, to make space for the new data. When an application tries to access an evicted page, an EPC fault is raised, and SGX brings the page back to the EPC [72].

An EPC fault is an expensive operation. SGX encrypts and calculates the MAC (encrypted hash) of a page prior to an eviction. When the page is brought back, it needs to be decrypted and integrity checked before its use. Our experiment found that evicting a page from the EPC takes on an average of 12,000 cycles.

2.3 Enclave Transitions: ECALLs and OCALLs

The security provided by SGX comes at a cost. For security reasons, SGX puts several restrictions on an executing enclave – notably, it cannot make any system calls [29].

In the Intel SGX framework, the operating system is an untrusted entity, and hence, systems calls are restricted. To make a system call, an enclave first exits the secure region by calling an OCALL (outside call) function. After this, it makes the system call, collects its results, and returns to the secure region. Similarly, an application from an unsecure region can call a function within an enclave by calling an ECALL (enclave call) function.

During a transition from the secure region to the unsecure region, the TLB entries of the enclave are flushed due to security concerns [29]. When the enclave returns, the TLB entries have to be populated again. While adding a TLB entry to the TLB, if the entry points to an EPC page, it is first verified. For this purpose, SGX maintains a table called the *Enclave Page Cache Map* or EPCM [29] in the secure region. The EPCM contains one entry for every page in the EPC. For each EPC page, the EPCM tracks its owner and the corresponding virtual address for which this page was allocated. These values are checked when a TLB entry for the corresponding page is being added to the TLB (see Figure 1).

Frequent enclave transitions affect the performance of an application due to context switches, TLB misses, and cache pollution. Weisse et al. [80] show that the cost of calling an enclave function typically requires 17,000 cycles.

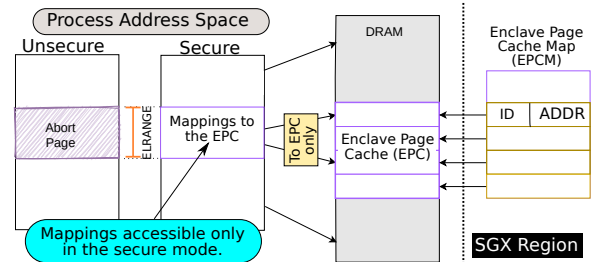


Figure 1: Figure showing the relation between the address space, the EPC, and the EPCM.

2.4 Library Operating Systems

Intel provides a software development kit (Intel SGX SDK [40]) to ease application development for SGX. However, porting or writing an application for SGX still requires an in-depth knowledge of the workings of SGX and significant development effort [36].

Due to this, many researchers have opted for executing an application using a shim layer built on top of Intel SGX– also known as a *library operating system* or LibOS. A library operating system can execute an unmodified binary on Intel SGX; thus, saving on the high cost and effort of porting the application. Scone [68], GrapheneSGX [24], and Panoply [70] are some examples of such systems. Researchers have reported that it can take up to 3-4 months for porting applications [36]; this is also in line with our observations. Moreover, the task of verifying correct execution for all corner cases will take even more time. This is precisely why such

shim layers need to be inevitably used and are fast becoming an inseparable part of the Intel SGX stack. Even though they have their share of performance overheads, the sheer reduction in the development and verification effort makes them a necessary part of many deployments. No benchmark suite for Intel SGX can be oblivious of them.

3 RELATED WORK AND MOTIVATION

In this section, we discuss the related work in this area and the motivation for SGXGauge.

3.1 Related Work

Limited work has been done in this area, mainly due to the limitations of the Intel SGX framework and the engineering effort required to port an application to SGX.

3.1.1 Lmbench-SGX. Hasan et al. [36] in their work *Port-or-Shim*, ported a part of Lmbench [56] for Intel SGX and compared its performance against a shimmed version running within a library operating system. They also used GrapheneSGX for their evaluations. They point out that porting Lmbench to SGX took months – and that too after removing certain features from it [36]. Whereas running a shimmed version of Lmbench on GrapheneSGX [24] took a week of effort. They specifically focus on the cost of the encryption/decryption and enclave transitions. They intentionally avoided EPC faults by ensuring that the amount of memory allocated to the benchmarks is less than the size of the EPC (92 MB). They report that the performance of Lmbench-SGX (ported version of Lmbench) and the version that runs within the library OS GrapheneSGX is the same – this raises a question about whether porting was worth the effort.

3.1.2 Nbench-SGX. Apart from this, Fu et al. [32] proposed a method to prevent side-channel attacks in Intel SGX. They ported Nbench [19] to SGX to evaluate the effectiveness of their solution. However, the working set of the benchmarks was small and limited analyses were performed.

Both Lmbench-SGX and Nbench-SGX (ported versions) are single-threaded benchmark suites [32, 36]. Lmbench-SGX mainly focuses on the memory bandwidth and the system call latencies. Nbench-SGX mostly contains CPU-intensive workloads and is designed to check the efficiency of integer and floating point operations of a CPU. Our suite is *far more comprehensive* in terms of its coverage (evaluated in Section 5).

Mahhouk et al. [53] also point out the issues with using Nbench-SGX for Intel SGX evaluation. They propose a CMake based framework to develop SGX and non-SGX applications from the same source. Apart from this, there are other proposals by Weichbrodt et al. [79], Krahn et al. [45], Bailleu et al. [13], and Lahey [48] that propose methods to collect statistics about an executing secure application. This information can help a developer debug a secure application or improve its performance. However, these are not benchmark suites; rather, they focus on the efficient profiling aspect for an executing enclave.

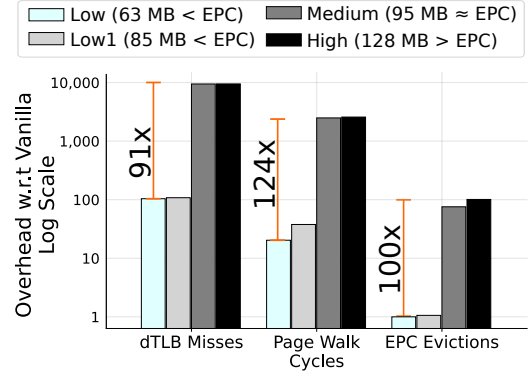


Figure 2: Allocating beyond the EPC size increases the overhead. The baseline is a *Vanilla* (non-SGX) setting with the same input size. For EPC evictions the baseline is the Low setting.

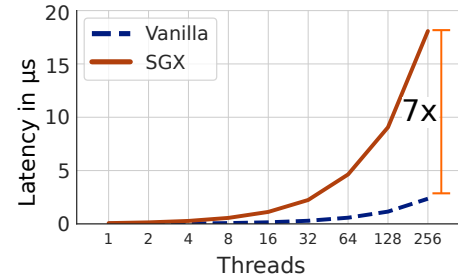


Figure 3: The latency of the *Lighttpd* server increases with the number of concurrent accesses by up to 7× while running in SGX and compared to a *Vanilla* (non-SGX) execution.

3.2 Motivation

Here, we discuss the motivation for SGXGauge. The system setup for experiments used here is listed in Table 3.

3.2.1 Experiment: Stressing the EPC. The limited amount of EPC memory is one of the biggest challenges in SGX [29, 51]. Due to the small size, EPC faults are a common event. Multiple instances of an enclave with a small memory footprint may also cause a number of EPC faults. This is because an enclave prior to its execution is loaded completely in the EPC to verify its content [29, 30].

As can be seen in Figure 2, on crossing the EPC boundary the number of dTLB misses increases by 91×, page walk cycles by more than 124×, and EPC evictions by 100× as compared to when the amount of memory is less than the EPC size. Hence, analyzing the impact of the EPC size on the performance is crucial – a fact completely ignored by Lmbench-SGX [36] and Nbench-SGX [32].

3.2.2 Experiment: Execution of multi-threaded benchmarks. An application leverages the multiple cores on a modern system by using threads to speed up its operation. Intel SGX does not support thread creation inside the secure region; however, numerous threads can do an ECALL and execute the same function using the same global enclave ID [39]. The overhead due to Intel SGX can change drastically based on the number of threads making

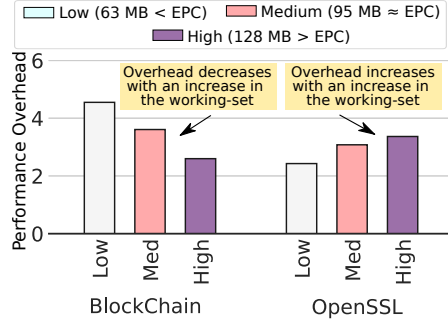


Figure 4: A library operating system may affect the performance of an application in a positive or negative manner, depending on the characteristics of the application.

the ECALL. As shown in Figure 3, the latency of Lighttpd increases with the number of threads (by 7×). Hence, it is crucial to capture the executions’ characteristics in this setting also. Nbench-SGX and Lmbench-SGX *do not contain* any multi-threaded benchmarks.

3.2.3 Experiment: Library Operating System. As noted in prior work [32, 36] and also by us, shimming an application is much easier than porting an application for Intel SGX— in terms of development and verification effort. We believe that in the future a library operating system will be the primary way to execute an application on Intel SGX. Hence, it is essential to understand the behavioral changes between ported and shimmed applications. *Port-or-Shim* [36] also focuses on this problem, but with benchmarks that have a small working set (70MB). Our observations are more comprehensive and also differ. As shown in Figure 4, the impact of a library operating system depends on the characteristics of the application and thus needs to be rigorously studied.

3.2.4 Experiment: Real-world benchmarks. Real world applications exhibit different phases during their execution. A typical pattern is that an application will read some data from the file system, process it, and then store the results. Micro-benchmarks such as Nbench [19] lack this phase change behavior and thus do not represent a real-world scenario (details in Section 5). Let us summarize.

- (1) Existing benchmark suites for Intel SGX [32, 36] are ported version of decade-old benchmarks that were designed to evaluate the CPU performance; they are not well suited to SGX.
- (2) Using multiple threads changes the overheads of Intel SGX, hence it is necessary to also include multi-threaded benchmarks.
- (3) The performance impact of a library operating system depends on the characteristics of the application executing within it. Thus, there is a need for further study.
- (4) Modern applications show different phases during their execution. Therefore, it is necessary to assess Intel SGX using real-world benchmarks.

4 SGXGAUGE BENCHMARK SUITE

The most important challenge in front of us was to find an appropriate set of workloads that need to be executed in a secure environment. This problem has a degree of subjectivity. Nonetheless, we followed standard practice and restricted ourselves to workloads that have been used by highly cited works on SGX in the recent past. We found the following workloads: blockchain related [12, 14, 15, 55, 78], protecting key-value pairs [25, 43, 44, 52], securing databases [34, 64, 77, 84], protecting keys [18, 21], securing a machine learning models [11, 42, 47], protecting network routing tables [63], securing communication [54], graph traversals [27, 41], protecting web-servers [68, 80], and HPC workloads [81].

The next task was to refine the set of workloads and choose an appropriate set. There are three main sources of overheads in Intel SGX: encryption/decryption, enclave transitions, and EPC faults (see Section 2). While selecting workloads for SGXGauge, our primary aim was to ensure complete coverage of all the Intel SGX components. First, we selected some of the most commonly used workloads such as OpenSSL [38, 61] and Lighttpd [24, 69, 80]. We then analyzed them and identified where they lack in terms of stressing the SGX components. For e.g. both OpenSSL and Lighttpd do not stress the CPU much. Hence, to fulfil this criterion, we selected Blockchain workload which is a CPU-intensive and multi-threaded workload. However, while it stresses the CPU, it does not use a lot of memory. To ensure both the components are stressed, we opted for an HPC workload XSBench, that was used by a prior work [81] for similar purposes.

For selecting workloads that exclusively stress the EPC, we selected the following from prior work: B-Tree, BFS, HashJoin, and PageRank. Each of them has different data access patterns. B-Tree is used commonly in databases for efficient lookups and has been used in Intel SGX setting also [34]. BFS is used in protecting the control flow graph of an application [27, 46]. While B-Tree aims to minimize the number of nodes it visits, BFS aims to visit all nodes efficiently. HashJoin performs a number of hash table probing operations which is at the core of many systems [54, 63]. PageRank [4] is a widely used workload for link analysis. SVM is a machine learning (ML) workload that is CPU and memory-intensive. It runs multiple iterations over the same input data, a typical pattern of ML workloads. We also discarded some workloads such as Redis [7], Fourier transform [23], License Managers [76], GUPS [58], Nginx [65], etc. because they were similar to other workloads that were already chosen.

4.1 Evaluation Modes and Input Settings

We execute the workloads in SGXGauge in different modes and different input settings to gain a better understanding of Intel SGX workings. Table 1 list the different execution modes and input setting used in the paper.

4.2 Workloads’ Description

4.2.1 Blockchain [59]: A blockchain is a distributed ledger that does not require any central authority for its management. A blockchain is essentially a linked list of blocks. A block has a payload portion (contents of the block) and the hash of contents of the previous block on the chain. Given that the chain is immutable,

Table 1: Conventions used in the paper for discussion

Execution Modes	
<i>Vanilla</i>	An application executing without Intel SGX support.
<i>Native</i>	An application executing within Intel SGX after it is ported to the SGX framework.
<i>LibOS</i>	An application executing with Intel SGX in shimmed mode – with the support of a LibOS (GrapheneSGX).
Input Settings	
Low	$Memory(\text{footprint}) < Size(EPC)$, wherever applicable
Medium	$Memory(\text{footprint}) \approx Size(EPC)$, wherever applicable
High	$Memory(\text{footprint}) > Size(EPC)$, wherever applicable

these hashes have a degree of finality and permanence. We vary the number of blocks in the chain to create different inputs for the workload (see Table 2). The hash computation is the sensitive operation; hence, this operation is offloaded to Intel SGX. This function is called by many threads from the unsecure region resulting in many ECALLs.

4.2.2 OpenSSL [38, 53]: OpenSSL is a library that provides access to cryptographic primitives to developers. Our workload using Intel SGX-SSL [38] reads encrypted data from an input file and decrypts it within SGX. Then, it performs a small compute-intensive task based on the content of the decrypted file. Finally, it encrypts the generated output and saves it in the untrusted filesystem. This workload stresses the mechanisms that copy data from the unsecure memory region to the EPC and the EPC if the input file size is more than the EPC size.

4.2.3 B-Tree [34]. The B-Tree data structure is used for an efficient organization of data, specifically in database management systems. This enables efficient lookup in large databases and applications of a similar nature – a crucial feature in today’s “big-data” world. This workload creates a B-Tree consisting of a certain number of elements and performs multiple *find* operations on a randomly generated set of keys. This workload is also designed to stress the EPC and the paging system.

4.2.4 HashJoin [54, 63]: The hash-join algorithm is used in modern databases to implement “equi-join” [75]. It has two phases: build and probe. Given two data tables, it first builds a hash table from the rows in the first table, and then probe it using the rows in the second table. We vary the size of the first table and, in effect, vary the memory and compute-intensive nature of the workload.

4.2.5 Breadth-First Search (BFS) [31, 41]: The workload is a port from the implementation of the well-known breadth-first search algorithm used in the Rodinia benchmark suite [23]. The input to the workload is an undirected graph. It first reads the input graph to the EPC and then traverses all the connected components in the graph. This is primarily memory and compute-intensive workload. The computation overhead is a function of the number of nodes in the graph; the degree is at least 3.

4.2.6 Page Rank [81]: PageRank is used to rank web pages based on the popularity of pages that point to it. The input to the workload is a connected directed graph represented in the adjacency list format with an out-degree of at least 1. The workload loads the graph into the EPC and builds an adjacency matrix of pages with a

default initial rank for all. The workload then uses the number of out links of the page, previous rank, and the weight of the out neighbor pages to assign a new rank. This is repeated a fixed number of times.

4.2.7 Memcached: [6, 68, 80]. Memcached is an in-memory key-value store. It is used in production servers to cache hot data in memory. We use the popular YCSB [28] workload to evaluate the performance of Memcached. YCSB first populates Memcached with a specified amount of data and then performs a specified set of (read or write) operations on those key-value pairs.

4.2.8 XSBench: [74, 81]. XSBench is a key computational kernel of the Monte Carlo neutron transport algorithm over a set of “nuclides” and “grid-points” [74]. We vary the number of grid points to generate different input sizes for the workload (see Table 2).

4.2.9 Lighttpd [17, 24, 80]: Lighttpd is a light-weight web server that is optimized for concurrent accesses. The server however runs on a single thread. Our workload hosts a web-page of size 20 KB (similar to [80]). We use the *ab* tool, which is a part of the Apache suite [1], to make a certain number of requests to the Lighttpd server using concurrent threads (see Table 2).

4.2.10 Support Vector Machine (SVM) [22]. SVM is a popular machine learning technique to classify the input data by projecting it into a higher dimensional space, and then using a linear combination of separating functions. We implemented SVM using *libSVM* [22], a library to implement SVM in C/C++ code.

4.3 Porting to Intel SGX

SGXGauge contains 10 benchmarks. We have ported 6 of these to execute natively on Intel SGX (*native*). The other 4 are real-world benchmarks, which are evaluated in *LibOS* mode using GrapheneSGX (details in Section 5). For these benchmarks, the engineering and verification effort in creating a native SGX port was prohibitive, and the benefits were not clear. Table 1 summarizes this information and also talks about the three execution settings (with different memory footprints): Low, Medium, and High.

While porting an application to Intel SGX, the ideal case is to run the entire application within an enclave. However, this is not always possible due to the restrictions imposed by Intel SGX. In this case, a crucial function is typically moved to the enclave and is accessed via an ECALL. This is the standard practice [50, 57].

We follow this approach while porting the applications. We completely ported OpenSSL, BFS, PageRank, B-Tree, and HashJoin to Intel SGX. However, Blockchain uses multiple threads to speed up the hash finding process. Intel SGX does not support the creation of threads within an enclave [39]. Nevertheless, multiple threads from the untrusted region can call the same ECALL function. Hence, for Blockchain, we moved the hash function inside Intel SGX; it is called from different threads from the main application which runs in the untrusted region.

4.4 Running on GrapheneSGX

To execute a binary on GrapheneSGX, we first need to define a “manifest” file. The manifest file contains the binary’s location, list of libraries required, and the required input files. The parameters

Table 2: Description of the workloads in SGXGauge along with the specific settings used in the paper.

S.No.	Workload	Vanilla mode	Native mode	LibOS mode	Property	Low (< EPC)	Medium (\approx EPC)	High (> EPC)
1.	Blockchain [59]	✓	✓	✓	CPU/ECALL-intensive	Blocks 3	Blocks 5	Blocks 8
2.	OpenSSL [61]	✓	✓	✓	Data-intensive	File Size 76 MB	File Size 88 MB	File Size 151 MB
3.	BTree [9]	✓	✓	✓	Data/CPU-intensive	Elements 1 M	Elements 1.5 M	Elements 2 M
4.	HashJoin [10]	✓	✓	✓	Data/CPU-intensive	Data Table Size 61 MB	Data Table Size 91 MB	Data Table Size 122 MB
5.	BFS [71]	✓	✓	✓	Data-intensive	Nodes 70 K Edges 909 K	Nodes 100 K Edges 1.3 M	Nodes 150 K Edges 1.9 M
6.	Pagerank [71]	✓	✓	✓	Data-intensive	Nodes 4500 Edges 10.1 M	Nodes 4750 Edges 11.2 M	Nodes 5000 Edges 12.5 M
7.	Memcached [6]	✓	✗	✓	Data/ECALL-intensive	Records: 50K Operations: 800K	Records: 100K Operations: 800K	Records: 200K Operations: 800K
8.	XSBench [74]	✓	✗	✓	CPU-intensive	Points: 53K Lookups: 100	Points: 88K Lookups: 100	Points: 768K Lookups: 100
9.	Lighttpd [17]	✓	✗	✓	ECALL-intensive	Requests: 50K Threads: 16	Requests: 60K Threads: 16	Requests: 70K Threads: 16
10.	SVM [37]	✓	✗	✓	Data/CPU-intensive	Rows 4000 Features 128	Rows 6000 Features 128	Rows 10000 Features 128

such as the enclave size and the threads to be used are also listed here. GrapheneSGX then processes this file and calculates the hash of all the required input files, which are then verified at the time of the execution.

5 EVALUATION

Here, we discuss the performance of workloads in SGXGauge under different execution modes and with different input settings (see Table 1). We focus on the insights that are not listed in prior work [30, 36, 53, 83], or where our observations differ from theirs.

Table 3: System configuration

Hardware Settings		
Xeon E-2186G CPU, 3.80 GHz		Disk: 1 TB (HDD)
CPUs: 1 Socket, 6 Cores, 2 HT		
DRAM: 32 GB	L1: 384 KB, L2: 1536 KB, L3: 12 MB	
System Settings		
Linux kernel: 5.9		ASLR: Off GCC: 9.3.0
DVFS: fixed frequency (performance)		Transparent Huge Pages: never
SGX Settings		
PRM: 128 MB	Driver: 2.11	SDK version: 2.13
GrapheneSGX Settings		
Enclave Size: 4 GB	Threads: 16	Internal Memory: 64 MB

5.1 Experimental Setup

The details of our evaluated system can be seen in Table 3. We use the GrapheneSGX library operating system [24] for our experiments in *LibOS* mode. Since GrapheneSGX is under constant development, we found that the performance of the code in the master branch is significantly better than its official release (v1.1). Hence, we used the code from the master branch of their GitHub repository¹.

5.1.1 Instrumenting Intel SGX. In order to instrument SGX-related events, we added instrumentation code directly to the Intel SGX driver code. This approach has also been used in prior work [45, 48]. We identified crucial functions within the driver code that are called

Table 4: Overhead in system-related events. Avg. value of EPC evictions is reported when compared with the *Vanilla* mode. The overhead refers to the performance overhead (run time).

Native Mode w.r.t Vanilla (6 workloads)						
	Over-head	dTLB misses	Walk Cycles	Stall cycles	LLC misses	EPC Evictions
Low	2.0×	8.38×	29.7×	2.5×	1.8×	21.5 K
Medium	3.0×	14.6×	57.0×	5.3×	2.0×	49.6 K
High	3.4×	17.48×	59.1×	6.4×	3.0×	79.6 K
LibOS Mode w.r.t Vanilla (10 workloads)						
Low	2.03×	40.6×	517×	114×	24×	796 K
Medium	3.13×	59.7×	724×	146×	18.5×	1,792 K
High	3.7×	44.0×	113×	12.7×	15.5×	2,255 K
LibOS Mode w.r.t Native (6 workloads)						
Low	1.03×	3.3×	5.1×	8.3×	9.3×	75×
Medium	1.03×	2.7×	4.0×	7.9×	9.2×	68×
High	0.9×	2.0×	3.0×	5.9×	7.2×	45×

during different SGX events, such as `sgx_do_fault()` (page fault handling function). Note that these functions do not execute within the secure world and thus can be easily instrumented. We report the latencies of SGX’s crucial functions in Appendix A.

5.2 Evaluation Plan

We take the following approach for evaluation.

- **Native mode performance:** We analyze the impact of Intel SGX on the applications executing natively on it for different input sizes.
- **LibOS mode performance:** We study the overheads that are introduced while executing on Intel SGX using a library operating system.
- **Native mode vs. LibOS mode:** We compare the performance of the *Native* and *LibOS* execution modes.

Table 2 shows an overview of the evaluation results. The geometric mean value is computed across at least 10 executions (seen to be enough).

¹Commit ID: adf6269218dfa80aed276d57121a98e7b13b0f4e

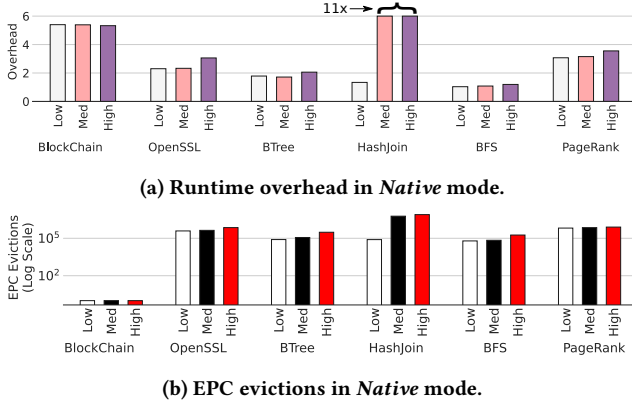


Figure 5: Performance impact of SGX on applications in *Native* mode for different input sizes.

5.3 Native mode Performance

Here, we evaluate the performance overhead of running an application in the *Native* mode as compared to the *Vanilla* mode with different input sizes. As shown in Figure 5a, the performance overhead increases by up to 8.8× as we go from the Low to the Medium setting, and by up to 1.4× from the Medium to the High setting.

As shown in Figure 5b, the total number of EPC evictions increase by up to 75× when the input size is increased from Low to Medium. On further increasing the input from Medium to High, the total number of EPC evictions increases by up to 2.6×. As already explained in Section 2, the TLB entries of an enclave are flushed before a transition to the unsecure region due to security reasons. Hence, as we increase the size from Low to Medium, dTLB misses increase by up to 79×, and by up to 2.6× as we go from Medium to High. Due to this, the total walk cycles increase by up to 79× (Low to Medium), and by up to 2.6× while going from Medium to High. Consequently, the total stall cycles increase by up to 43× (Low to Medium), and by up to 5.4× while going from Medium to High.

Summary: As we approach the EPC size (Low to Medium), there is a sudden rise in all the paging and TLB-related performance counters. However, going beyond the EPC size (Medium to High) does not affect the performance to that extent. We present a detailed discussion for all the workloads in Appendix B and impact of the counters on the workloads’ performance in Appendix C.

5.4 LibOS Mode Performance

Here, we evaluate the performance impact of GrapheneSGX.

5.4.1 GrapheneSGX Overhead. We first characterize the overhead of just GrapheneSGX using an “empty” (return 0;) workload. As shown in Figure 6a, in this setup, GrapheneSGX performs ≈300 ECALLs, ≈1000 OCALLs, and ≈1000 AEX exits. During this time, total EPC evictions are ≈ 1 M. However, out of these 1 M evicted EPC pages, only ≈ 700 pages (2 MB) are loaded back.

The reason for the unusually high number of EPC evictions is the enclave size property, which is set to 4 GB. As prior to executing an enclave, SGX completely loads it in the EPC to calculate its signature [30]. Doing so for a 4 GB enclave will cause 1 M EPC faults

(1 M*4 KB=4 GB). Lowering the value of the property “enclave-size” reduces the EPC evictions but worsens the performance by up to 4×, even for the workloads with a small memory footprint such as Blockchain. All these EPC evictions are done at the beginning of the execution, i.e., while initializing GrapheneSGX. We do not count this time in the execution time of a workload running on it (see Appendix D).

Performance As shown in Figure 6b, the performance overhead increases by up to 8.7× while going from Low to Medium, and by up to 2.7× while going from Medium to High. As shown in Figure 6c, the total number of EPC load-backs (page brought back to the EPC from the untrusted memory) increases by up to 341× when the input size increases from Low to Medium, and by up to 4.1× on further increasing the input size from Medium to High. The total number of dTLB misses increases by up to 18.6× as we go from Low to Medium, and by up to 1.5× as we go from Medium to High. Due to this, the total number of walk cycles increases by up to 11× as we go from Low to Medium, and by up to 2.7× when we go from Medium to High. Consequently, the total number of stall cycles increases by up to 5.11× and 2.17×, respectively.

Summary: Similar to *Native* mode, as we approach the EPC size (Low to Medium), there is a sudden increase in all the TLB and paging-related performance counter values. However, going beyond the EPC size (Medium to High) does not affect the performance as much. We discuss the I/O related overheads with GrapheneSGX in Appendix E.

5.5 Native Mode vs LibOS Mode

Here, we compare the performance of the *Native* and *LibOS* modes (see Table ??). We observe that as we increase the input size, the performance overhead of *LibOS* as compared to the *Native* mode starts decreasing. The overall overhead reduces by 12.6% when we increase the input setting from Low to High or Medium to High. The total number of dTLB misses comes down by 18% and 25%, when the input is increased from Low to Medium and Medium to High, respectively. In the same setting, the total walk cycles comes down by 21% and 25%, stall cycles by 4% and 25%, LLC misses by 1% and 21%, and EPC evictions by 9% and 33% when we increase the input size from Low to Medium and Medium to High, respectively.

Summary: On increasing the workload size, the overhead of GrapheneSGX starts decreasing, and eventually, approaches that of the *Native* mode.

5.6 Switchless Mode

To reduce the cost of an OCALL, Intel SGX supports a switchless mode of operation where it leverages multiple cores of a modern system to make an OCALL without exiting an enclave – thus preventing a TLB flush. In this case, a set of threads (proxy threads) on dedicated cores are used to handle the OCALLs. Here, the parameters of an OCALL and other relevant data are sent to a *proxy thread* running on another core using an unsecure shared memory channel. The proxy thread reads the request and performs the operation. Once the operation is finished, the results are written to the shared memory region; these results are subsequently read by the enclave that issued the request. This is a standard pattern and is used to

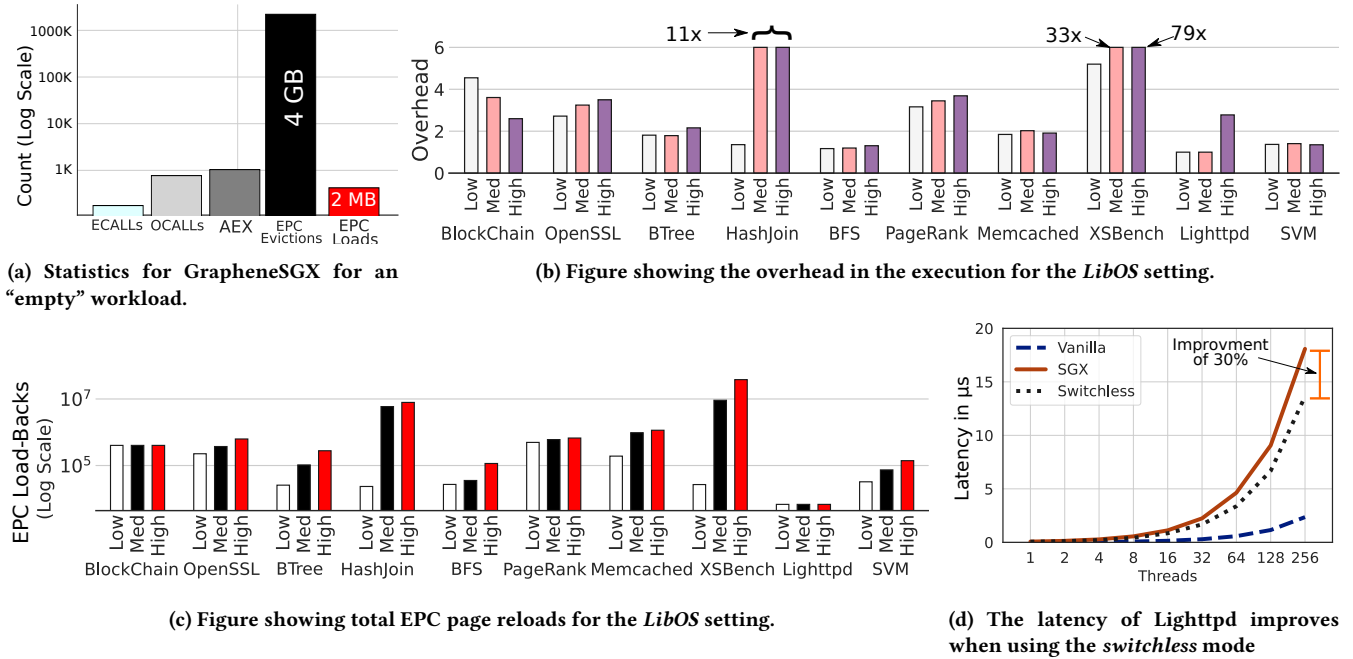


Figure 6: Performance impact of GrapheneSGX on workloads in SGXGauge.

to hide the overheads of system calls as well in regular operating systems.

We configured GrapheneSGX to use 8 cores for handling OCALL requests from enclaves. In Lighttpd, this reduces the total number of dTLB misses by 60% thus improving the latency by 30%, as compared to the default implementation of OCALL (see Figure 6d).

6 CONCLUSION

We introduced SGXGauge, a benchmark suite for Intel SGX that captures a holistic view of the performance of applications running in such TEEs – this includes the impact of the EPC memory. SGXGauge contains diverse benchmarks that affect different components of SGX. We also performed an evaluation of the performance of SGX in LibOS mode and showed that there is a marked difference in behavior as the memory footprint crosses the EPC size limit.

REFERENCES

- [1] [n.d.]. ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4. <https://http.apache.org/docs/2.4/programs/ab.html>. (Accessed on 06/21/2021).
- [2] [n.d.]. Intel Launches Its Most Advanced Performance Data Center Platform | Business Wire. <https://www.businesswire.com/news/home/20210406005302/en/Intel-Launches-Its-Most-Advanced-Performance-Data-Center-Platform>. (Accessed on 06/15/2021).
- [3] [n.d.]. Manifest syntax — Graphene documentation. <https://graphene.readthedocs.io/en/latest/manifest-syntax.html>. (Accessed on 05/26/2021).
- [4] [n.d.]. PageRank - Wikipedia. <https://en.wikipedia.org/wiki/PageRank>. (Accessed on 06/24/2021).
- [5] 2019. Academic Research | Intel Software Guard Extensions | Intel Software. <https://software.intel.com/en-us/sgx/documentation/academic-research>. (Accessed on 11/18/2019).
- [6] 2019. memcached - a distributed memory object caching system. <https://memcached.org/>. (Accessed on 11/18/2019).
- [7] 2019. Redis. <https://redis.io/>. (Accessed on 11/18/2019).
- [8] 2020. RDTSC — Read Time-Stamp Counter. <https://www.felixcloutier.com/x86/rdtsc>. (Accessed on 05/26/2020).
- [9] Reto Achermann. 2020. mitosis-project/mitosis-workload-btree: The BTree workload used for evaluation. <https://github.com/mitosis-project/mitosis-workload-btree>. (Accessed on 10/03/2020).
- [10] Reto Achermann. 2020. mitosis-project/mitosis-workload-hashjoin: The HashJoin workload used for evaluation. <https://github.com/mitosis-project/mitosis-workload-hashjoin>. (Accessed on 10/03/2020).
- [11] Dávid Ács and Adrian Colea. 2019. Securely Exposing Machine Learning Models to Web Clients using Intel SGX. 2019 IEEE 15th International Conference on Intelligent Computer Communication and Processing (ICCP) (2019), 161–168.
- [12] Gbadebo Ayoade, Vishal Karande, Latifur Khan, and Kevin Hamlen. 2018. Decentralized IoT Data Management Using Blockchain and Trusted Execution Environment. In 2018 IEEE International Conference on Information Reuse and Integration (IRI). 15–22. <https://doi.org/10.1109/IRI.2018.00011>
- [13] Maurice Bailleu, Donald Dragoti, Pramod Bhatotia, and Christof Fetzter. 2019. TEE-Perf: A Profiler for Trusted Execution Environments. In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 414–421. <https://doi.org/10.1109/DSN.2019.00050>
- [14] Zijian Bao, Qinghao Wang, Wenbo Shi, L. Wang, Hong Lei, and Bangdao Chen. 2020. When Blockchain Meets SGX: An Overview, Challenges, and Open Issues. IEEE Access 8 (2020), 170404–170420.
- [15] G. Bashar, Alejandro Anzola Avila, and Gaby G. Dagher. 2020. PoQ: A Consensus Protocol for Private Blockchains Using Intel SGX. In SecureComm.

- [16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) (PACT '08). Association for Computing Machinery, New York, NY, USA, 72–81. <https://doi.org/10.1145/1454115.1454128>
- [17] Andre Bogus. 2008. *Lighttpd installing, compiling, configuring, optimizing, and securing this lightning-fast Web Server* / Andre Bogus. Packt Publishing, Birmingham, UK.
- [18] Helena Brekalo, Raoul Strackx, and F. Piessens. 2016. Mitigating Password Database Breaches with Intel SGX. *Proceedings of the 1st Workshop on System Software for Trusted Execution* (2016).
- [19] BYTE. 1995. NBench. <https://www.math.utah.edu/~mayer/linux/bmark.html>. (Accessed on 09/23/2019).
- [20] Don Capps et al. 2019. IOzone Filesystem Benchmark. <http://www.iozone.org/>. (Accessed on 11/15/2019).
- [21] Somnath Chakrabarti, B. Baker, and Mona Vij. 2017. Intel SGX Enabled Key Manager Service with OpenStack Barbican. *ArXiv abs/1712.07694* (2017).
- [22] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 3, Article 27 (May 2011), 27 pages. <https://doi.org/10.1145/1961189.1961199>
- [23] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing (IISWC '09). IEEE Computer Society, USA, 44–54. <https://doi.org/10.1109/IISWC.2009.5306797>
- [24] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference*.
- [25] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. 2019. EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX (Middleware '19). Association for Computing Machinery, New York, NY, USA, 14–27. <https://doi.org/10.1145/3361525.3361533>
- [26] S. Chen, A. Ailamaki, P.B. Gibbons, and T.C. Mowry. 2004. Improving hash join performance through prefetching. In *Proceedings. 20th International Conference on Data Engineering*. 116–127. <https://doi.org/10.1109/ICDE.2004.1319989>
- [27] Joseph I. Choi, Dave (Jing) Tian, Grant Hernandez, Christopher Patton, Benjamin Mood, Thomas Shrimpton, Kevin R. B. Butler, and Patrick Traynor. 2019. A Hybrid Approach to Secure Function Evaluation Using SGX. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland, New Zealand) (Asia CCS '19). Association for Computing Machinery, New York, NY, USA, 100–113. <https://doi.org/10.1145/3321705.3329835>
- [28] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) (SoCC '10). ACM, New York, NY, USA, 143–154.
- [29] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.
- [30] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. 2019. Everything You Should Know About Intel SGX Performance on Virtualized Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 5 (March 2019), 21 pages. <https://doi.org/10.1145/3322205.3311076>
- [31] David Erick Lavoie. [n.d.]. Sable/bfs-benchmark. <https://github.com/Sable/bfs-benchmark>. (Accessed on 06/19/2021).
- [32] Yangchun Fu, Erick Bauman, Raul Quinonez, and Z. Lin. 2017. Sgx-Lapd: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *RAID*.
- [33] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *ACM Comput. Surv.* 51, 2, Article 26 (March 2018), 33 pages. <https://doi.org/10.1145/3158644>
- [34] A. Gribov, Dhinakaran Vinayagamurthy, and S. Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proceedings on Privacy Enhancing Technologies* 2019 (2019), 370 – 388.
- [35] Ahmed A. Harby, S. Fahmy, and A. F. Amin. 2019. More Accurate Estimation of Working Set Size in Virtual Machines. *IEEE Access* 7 (2019), 94039–94047.
- [36] Aisha Hasan, Ryan Riley, and Dmitry Ponomarev. 2020. Port or Shim? Stress Testing Application Performance on Intel SGX. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 123–133. <https://doi.org/10.1109/IISWC50251.2020.00021>
- [37] Marti A. Hearst. 1998. Support Vector Machines. *IEEE Intelligent Systems* 13, 4 (July 1998), 18–28. <https://doi.org/10.1109/5254.708428>
- [38] Intel. [n.d.]. intel/intel-sgx-ssl: Intel® Software Guard Extensions SSL. <https://github.com/intel/intel-sgx-ssl>. (Accessed on 06/23/2021).
- [39] Intel. [n.d.]. Intel(R) Software Guard Extensions SDK Developer Reference for Linux® OS. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf. (Accessed on 06/28/2021).
- [40] Intel. 2019. Intel SGX for Linux*. <https://github.com/intel/linux-sgx>. (Accessed on 09/23/2019).
- [41] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs With SGX (ASIA CCS '17). Association for Computing Machinery, New York, NY, USA, 19–30. <https://doi.org/10.1145/3052973.3053034>
- [42] Ryan Karl, Jonathan Takeshita, and Taeho Jung. 2020. Using Intel SGX to improve private neural network training and inference. *Proceedings of the 7th Symposium on Hot Topics in the Science of Security* (2020).
- [43] Taehoon Kim, J. Park, J. Chang, Seungheun Jeon, and Jaehyuk Huh. 2018. Secure In-memory Key-Value Storage with SGX. *Proceedings of the ACM Symposium on Cloud Computing* (2018).
- [44] Taehoon Kim, J. Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. *Proceedings of the Fourteenth EuroSys Conference 2019* (2019).
- [45] Robert Krahn, Donald Dragoti, Franz Gregor, Do Le Quoc, Valerio Schiavoni, Pascal Felber, Clenimar Souza, Andrey Brito, and Christof Fetzer. 2020. TEEMon: A Continuous Performance Monitoring Framework for TEEs. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) (Middleware '20). Association for Computing Machinery, New York, NY, USA, 178–192. <https://doi.org/10.1145/3423211.3425677>
- [46] Sandeep Kumar, Diksha Moolchandani, T Ono, and Smruti R Sarangi. 2019. F-LaaS: A Control-Flow-Attack Immune License-as-a-Service Model. In *2019 IEEE International Conference on Services Computing (SCC)*. 80–89.
- [47] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnaudov, Pramod Bhatotia, and Christof Fetzer. 2019. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. (2019). arXiv:1902.04413. <http://arxiv.org/abs/1902.04413>
- [48] Kevin Lahey. 2020. Monitoring Intel® SGX Enclaves. <https://fortanix.com/blog/2020/02/monitoring-intel-sgx-enclaves/>. (Accessed on 05/12/2021).
- [49] Ankur Limaye and Tosiron Adegbiya. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 149–158. <https://doi.org/10.1109/ISPASS.2018.00028>
- [50] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Evers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference*.
- [51] Ximing Liu, Wenwen Wang, Lizhi Wang, Xiaoli Gong, Ziyi Zhao, and Pen-Chung Yew. 2020. Regaining Lost Seconds: Efficient Page Preloading for SGX Enclaves. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) (Middleware '20). Association for Computing Machinery, New York, NY, USA, 326–340. <https://doi.org/10.1145/3423211.3425673>
- [52] D. Lucani, Marcell Fehér, K. Fonseca, M. Rosa, and Bogdan Despotov. 2018. Secure and Scalable Key Value Storage for Managing Big Data in Smart Cities Using Intel SGX. *2018 IEEE International Conference on Smart Cloud (SmartCloud)* (2018), 70–76.
- [53] Mohammad Mahhouk, Nico Weichbrodt, and Rüdiger Kapitza. 2021. SGX-oMeter: Open and Modular Benchmarking for Intel SGX. In *Proceedings of the 14th European Workshop on Systems Security* (Online, United Kingdom) (EuroSec '21). Association for Computing Machinery, New York, NY, USA, 55–61. <https://doi.org/10.1145/3447852.3458722>
- [54] Moxie Marlinspike. 2017. Signal » Blog » Technology preview: Private contact discovery for Signal. <https://signal.org/blog/private-contact-discovery/>. (Accessed on 06/24/2021).
- [55] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiaainen, Ghassan Karame, and Srđjan Capkun. 2019. BITE: Bitcoin Lightweight Client Privacy using Trusted Execution. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 783–800. <https://www.usenix.org/conference/usenixsecurity19/presentation/matetic>
- [56] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (San Diego, CA) (ATEC '96). USENIX Association, USA, 23.
- [57] M. Melara, M. Freedman, and M. Bowman. 2019. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. *ArXiv abs/1907.13245* (2019).
- [58] Alex Merritt. [n.d.]. alexandermerritt/gups: gups mirror. <https://github.com/alexandermerritt/gups>. (Accessed on 06/23/2021).
- [59] Saurav Mohapatra. 2019. mohaps/libcatena: a simple toy blockchain written in C++ for learning purposes. <https://github.com/mohaps/libcatena>. (Accessed on 09/23/2019).
- [60] Vlad Nitu, A. Kocharyan, Hannas Yaya, A. Tchana, D. Hagimont, and H. Astsatryan. 2018. Working Set Size Estimation Techniques in Virtualized Environments: One Size Does not Fit All. *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems* (2018).
- [61] OpenSSL. 2019. OpenSSL. <https://www.openssl.org/>. (Accessed on 12/07/2019).
- [62] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves (EuroSys '17). Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3064176.3064219>
- [63] Nicolae Paladi, Jakob Svenningsson, Jorge Medina, and Patrik Arlos. 2019. Protecting OpenFlow Flow Tables with Intel SGX (SIGCOMM Posters and Demos '19). Association for Computing Machinery, New York, NY, USA, 146–147.

- <https://doi.org/10.1145/3342280.3342339>
- [64] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. *2018 IEEE Symposium on Security and Privacy (SP)* (2018), 264–278.
 - [65] Will Reese. 2008. Nginx: The High-Performance Web Server and Reverse Proxy. *Linux J.* 2008, 173, Article 2 (Sept. 2008).
 - [66] Colin Robertson. [n.d.]. `_rdtsc` | Microsoft Docs. <https://docs.microsoft.com/en-us/cpp/intrinsics/rdtsc?view=msvc-160>. (Accessed on 04/02/2021).
 - [67] Mark Russinovich. [n.d.]. Azure and Intel commit to delivering next generation confidential computing | Azure Blog and Updates | Microsoft Azure. <https://azure.microsoft.com/en-in/blog/azure-and-intel-commit-to-delivering-next-generation-confidential-computing/>. (Accessed on 06/15/2021).
 - [68] Sergei Arnaudov and Bohdan Trach and Franz Gregor and Thomas Knauth and Andre Martin and Christian Priebe and Joshua Lind and Divya Muthukumaran and Dan O’Keeffe and Mark L. Stillwell and David Goltzsche and Dave Evers and Rüdiger Kapitza and Peter Pietzuch and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 16*. USENIX Association, Savannah, GA, 689–703. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov>
 - [69] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX (ASPLOS ’20). Association for Computing Machinery, New York, NY, USA.
 - [70] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *NDSS*.
 - [71] Julian Shun and Guy E. Blelloch. 2013. Ligma: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. <https://doi.org/10.1145/2517327.2442530>
 - [72] Meysam Taassori and Ali Sha. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’18)* (2018), 665–678.
 - [73] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yaviv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (Toronto, Canada) (SysTEX ’18)*. Association for Computing Machinery, New York, NY, USA, 22–27. <https://doi.org/10.1145/3268935.3268942>
 - [74] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. <https://www.mcs.anl.gov/papers/P5064-0114.pdf>
 - [75] Marios Trivizas. 2016. How we made joins 23 thousand times faster: part one | Lab notes. <https://crate.io/a/joins-faster-part-one/>. (Accessed on 06/24/2021).
 - [76] Peter Verhas. 2019. License3j: Free Licence Management Library. <https://github.com/verhas/License3j>. (Accessed on 11/18/2019).
 - [77] Yongzhi Wang, Lingtong Liu, Cuicui Su, Jiawen Ma, Lei Wang, Yibo Yang, Y. Shen, G. Li, Tao Zhang, and Xuwen Dong. 2017. CryptSQLite: Protecting Data Confidentiality of SQLite with Intel SGX. *2017 International Conference on Networking and Network Applications (NaNA)* (2017), 303–308.
 - [78] Yichuan Wang, Bing Ma, Tong Zhang, W. Gao, Yifan Ding, and Xinyu Jiang. 2020. Bottom-up: A SGX-based Blockchain Trusted Startup and Verification Scheme. *Proceedings of the 1st ACM International Workshop on Security and Safety for Intelligent Cyber-Physical Systems* (2020).
 - [79] Nico Weichbrodt, Pierre Louis Aublin, and Rüdiger Kapitza. 2018. SGX-Perf: A performance analysis tool for Intel SGX enclaves. *Proceedings of the 19th International Middleware Conference, Middleware 2018* (2018), 201–213.
 - [80] Ofir Weiss, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 81–93.
 - [81] Sujay Yadalam, V. Ganapathy, and Arkaprava Basu. 2021. SGXL: Security and Performance for Enclaves Using Large Pages. *ACM Trans. Archit. Code Optim.* 18 (2021), 12:1–12:25.
 - [82] Xin Yan and Xiao Gang Su. 2009. *Linear Regression Analysis: Theory and Computing*. World Scientific Publishing Co., Inc., USA.
 - [83] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. 2016. On the Performance of Intel SGX. In *2016 13th Web Information Systems and Applications Conference (WISA)*. IEEE, 184–187. <http://ieeexplore.ieee.org/document/7878255/>
 - [84] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-based Verifiable Database. *Proceedings of the 2021 International Conference on Management of Data* (2021).

A INTEL SGX LATENCIES

Instrumenting an application running in the secure mode within SGX is a non-trivial task due to the constraints imposed by Intel SGX. RDTSC instructions [8, 66], which are generally used to measure the cycles taken by an operation, are not allowed in SGX. Fortunately, the Intel SGX driver code does not execute in Intel SGX and thus can be easily instrumented.

We measured the latencies of the core Intel SGX operations: allocating a page (`sgx_alloc_page()`), evicting a page (`sgx_ewb()`), loading back a page (`sgx_eldu()`), and handling a page fault in SGX (`sgx_do_fault()`). We use the *ftrace* tool [33] for this purpose. SGX uses the EWB instruction to evict a page from the EPC and the ELDU instruction to load it back. While evicting an EPC page, first its MAC is calculated and then the page is encrypted. While loading back, the page is decrypted and its integrity is checked using the MAC [72].

These functions are highly optimized with latencies in the range of a few micro-seconds. Figure 7 reports the mean of 40K+ samples. The latency of evicting an EPC page is 16% more than loading back an EPC page. SGX evicts pages in a batch that is typically 16 pages. However, during a fault, a single page is loaded back.

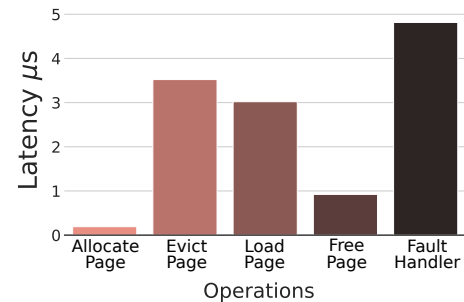


Figure 7: Latency of core operations in Intel SGX.

B NATIVE MODE PERFORMANCE

Here, we discuss the overheads in the 6 of workloads from SGX-Gauge while executing in the *Native* mode. We use the change in the most relevant hardware performance counters for this discussion. A heat-map of these counters is shown in Figure 8.

B.1 Blockchain

As seen in Figure 8a, while executing in the *Native* mode, the total number of dTLB misses is $\approx 2000\times$ more than the *Vanilla* mode. This is because in our implementation of Blockchain the hash function is protected inside an enclave. This is called via ECALLs from many threads that are executing in the unsecure region. This results in many enclave transitions and thus many TLB flushes. These TLB entries are to be populated for every ECALL by a page table walk. Hence, we see a similar increase in the number of walk cycles.

With 16 threads in the Low setting there are $\approx 3,133$ K ECALLs, for Medium $\approx 4,831$ K ECALLs, and for the High setting there are $\approx 8,944$ K ECALLs.

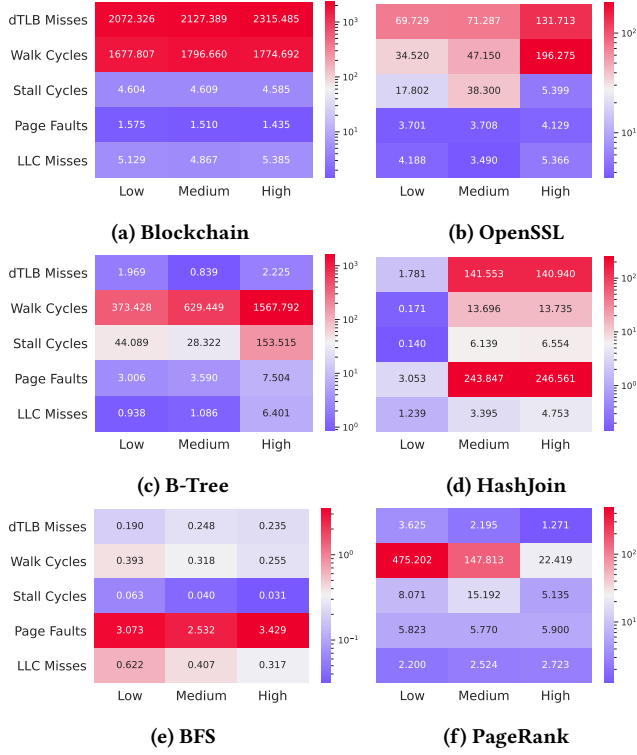


Figure 8: Overheads for the workloads when executing in the Native mode w.r.t the Vanilla mode.

B.2 OpenSSL

In OpenSSL, the number of EPC evictions increases from 389 K to 433 K to 721 K, as we increase the input size from Low to Medium and High, respectively. Due to this, in the High setting, the total number of enclave exits increases, thus increasing the total number of dTLB misses by 131 \times and walk cycles by 196 \times w.r.t. the *Vanilla* mode.

B.3 B-Tree

In B-Tree, the total number of EPC evictions increases from 79 K to 116 K to 305 K, when we move from the Low to Medium and then to the High input setting, respectively. However, here the total number of dTLB misses increase by only 2.2 \times only in the High setting. This is because the total number of dTLB misses is dominated by the number of page faults caused by the workload. To serve a page fault, an enclave performs an asynchronous exit (AEX), which also causes a TLB flush. The total number of page faults increases from 3 \times in the Low setting to 7.5 \times in the High setting, and the total LLC misses increase from ≈ 1 in the Low setting to ≈ 6.4 in the High setting.

B.4 HashJoin

In HashJoin, on increasing the input size we observe an increase in almost all of the performance counters. Most notably, the total number of page faults and dTLB misses increase by $\approx 246\times$ and $\approx 140\times$ over *Vanilla* mode in the High input setting, respectively.

This is due to the characteristics of the workload. A typical hashjoin operation incurs many cache misses and stall cycles [26].

B.5 BFS

In BFS, the total number of page faults increase by 3 \times as compared to the *Vanilla* mode. However, we do not observe a large impact with the increase in the input size. This is because of the inherent locality in the workload.

B.6 PageRank

In PageRank, we observe a decrease in the total number of walk cycles on increasing the input size. This is because dTLB misses also go down with an increasing input size. The main reason for this is the nature of the workload. In the *Vanilla* mode (not shown in the Figure), the number of dTLB misses increases by 3.6 \times when we increase the input size from the Low to the High setting. Hence, the nature of the workload dominates the total number of misses, hiding the extra misses caused due to SGX.

C COUNTER IMPACT ON PERFORMANCE

Intel SGX provides a way to execute an application securely on a remote machine, although with some limitations. Researchers are working on developing methods to circumvent these limitations. Different solutions might affect the components of Intel SGX differently. Here we provide a generic approach for the developer to select correct benchmarks from SGXGauge as per the requirement.

As pointed out in prior work and observed by us in our experiments, when a benchmark reserves more memory than the EPC size it suffers a slowdown. Usage of more memory than the EPC may impact the total number of dTLB misses, LLC misses, walk cycles, stall cycles, and EPC-Evictions. We rank the metrics in the order of importance for each of the workloads in SGXGauge. We use linear regression [82] for this purpose. Linear regression predicts the execution time given these metrics as input. While doing so, it assigns coefficients to these metrics. The magnitude of these coefficients is correlated with the importance of that metric in determining the execution time (see Table 5).

Table 5: Table showing the most important hardware performance counter that determines the performance of each workload (shown in bold). LLC refers to the last level cache.

Workloads	Walk cycles	Stall cycles	Page faults	dTLB misses	LLC misses	EPC evictions
<i>Native mode</i>						
Blockchain	0.33	0.32	0.01	0	0.32	0
OpenSSL	0.08	0.12	0.17	0	0.21	0.14
BTree	0.27	-0.11	0.22	-0.05	0.22	0.11
HashJoin	0.21	0.16	0.21	-0.03	0.21	0.21
BFS	0.10	0.17	0.18	0.21	0.09	0.22
PageRank	0.44	-0.54	0.05	0.33	0.65	0.04
<i>LibOS mode</i>						
Memcached	0.03	0.04	0.09	0.15	0.13	0.09
XSBench	0.16	0.17	0.17	0.18	0.16	0.17
Lighttpd	0.18	0.19	0	0.09	0.26	0
SVM	0.09	0.60	0.27	0.09	0.31	-0.03

We can conclude that most of the time paging and TLB-related counters are the most correlated with the performance. LLC misses are mostly an important factor in OpenSSL.

D GRAPHENESGX START-UP OVERHEAD

Here, we discuss the overhead in initializing GrapheneSGX.

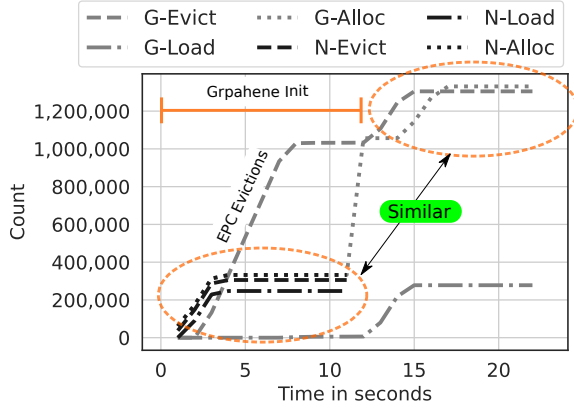


Figure 9: Figure showing the performance counter values for EPC page allocation, eviction, and load-back during the execution phase of B-Tree in the *Native* mode (N-) and *LibOS* mode (G-).

Intel SGX verifies the signature of an enclave prior to its execution. To do so, it loads the entire enclave into the EPC. In SGX v1, a heap size greater than the EPC size was not allowed, as that will not allow SGX to load the complete enclave in the EPC. Since SGX v2, a heap size greater than the EPC is allowed. SGX transparently evicts and loads pages as per its requirement.

Figure 9 shows the allocation, eviction, and load back of EPC pages in *Native* and *LibOS* modes for a representative workload, B-Tree. This pattern remains the same across other workloads also. SGX first calculates the signature of the enclave that causes the initial EPC evictions. Note that EPC pages are allocated after the verification is done. After that, the EPC pattern of GrapheneSGX is the same as that of the *Native* mode.

Intel SGX recommends setting the enclave size as per the maximum requirement of the application. However, in our experiments we observed additional overheads on setting a lower value of the enclave size in *LibOS* mode. This is related to how GrapheneSGX initializes the enclave. We thus used an enclave size of 4 GB for all our experiments. We do not count the GrapheneSGX start up time in the workload executing time while calculating the overheads in Section 5 mainly because this is a one-time activity and a workload can run for a very long time after its enclave is initialized. Also note in Figure 9 that after the initialization phase the gray (GrapheneSGX) and black (*Native*) lines converge (same behavior).

E WHAT ABOUT I/O?

As mentioned before, SGX does not support system calls, notably file system calls. An enclave needs to rely on an OCALL to read or write a file to the file system. In this case, by default, the data is transferred in plaintext and it is the responsibility of the developer to protect it via encryption. SGX has a sealing feature, where the data can be encrypted using the *sealing* enclave [29]. The sealing enclave is an Intel-authored enclave that is part of the Intel SDK. It

can “seal” or encrypt data using a platform dependent hardware key. The sealed data can only be “unsealed” or decrypted on the same platform, and optionally, it can be configured to be decrypted only by the same enclave that encrypted it.

Library operating systems support file system operations by transparently capturing these calls and handling them either via an OCALL or via a parallel, proxy thread executing on a different core. However, as this is not covered by Intel SGX constructs, a naive implementation will still write the data in plain text to the file system, essentially leaking data. GrapheneSGX supports transparently encrypting files before they are written to the file system. This feature is known as the protected file system or PF [3] mode. However, as shown in Figure 10 this feature is not optimized, and the performance of an I/O intensive application can suffer by up to 98% when PF is used.

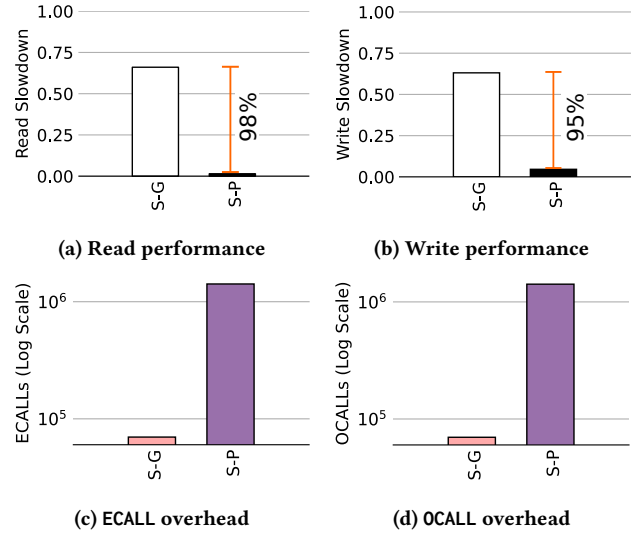


Figure 10: The I/O overhead with GrapheneSGX (S-G) and GrapheneSGX with protected files (S-P). Iozone: reading and writing 1 GB of data with 4 M blocks.

We use the popular file system benchmark Iozone [20] to evaluate the performance of the GrapheneSGX PF system. We compare this against the *Vanilla* mode, and *LibOS* mode without using the protected file setting. *LibOS* incurs an overhead of 33% and 36% compared to the *Vanilla* mode for read and write operations, respectively. The overhead increases to 98% and 95% for read and write operations, respectively, when the protected files mode is enabled. The main reason for this is the increase in the number of ECALLs (see Figure 10c) and OCALLs (see Figure 10d).

The PF mode needs to be optimized to make it practical for production-quality systems.