

SGX Switchless Calls Made Configless

Peterson Yuhala
University of Neuchâtel
Neuchâtel, Switzerland
peterson.yuhala@unine.ch

Michael Paper
ENS de Lyon
Lyon, France
michael.paper@ens-lyon.fr

Timothée Zerbib
Institut Polytechnique de Paris
Paris, France
timothee.zerbib@ip-paris.fr

Pascal Felber
University of Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

Valerio Schiavoni
University of Neuchâtel
Neuchâtel, Switzerland
valerio.schiavoni@unine.ch

Alain Tchana
Grenoble INP
Grenoble, France
alain.tchana@grenoble-inp.fr

Abstract—Intel’s software guard extensions (SGX) provide hardware enclaves to guarantee confidentiality and integrity for sensitive code and data. However, systems leveraging such security mechanisms must often pay high performance overheads. A major source of this overhead is SGX enclave transitions which induce expensive cross-enclave context switches. The Intel SGX SDK mitigates this with a *switchless* call mechanism for transitionless cross-enclave calls using worker threads. Intel’s SGX switchless call implementation improves performance but provides limited flexibility: developers need to statically fix the system configuration at build time, which is error-prone and misconfigurations lead to performance degradations and waste of CPU resources. ZC-SWITCHLESS is a configless and efficient technique to drive the execution of SGX switchless calls. Its dynamic approach optimises the total switchless worker threads at runtime to minimise CPU waste. The experimental evaluation shows that ZC-SWITCHLESS obviates the performance penalty of misconfigured switchless systems while minimising CPU waste.

Index Terms—Intel SGX, trusted execution environments, SGX switchless calls, multithreading

I. INTRODUCTION

Cloud computing increases privacy concerns for applications offloaded to the cloud, as sensitive user data and code are largely exposed to potentially malicious cloud services. Trusted execution environments (TEEs) like Intel SGX [3], [9], [11], [19] provide secure *enclaves*, offering confidentiality and integrity guarantees to sensitive code and data. Enclaves cannot be accessed by privileged or compromised software stacks, including the operating system (OS) or hypervisor. However, SGX enclaves introduce overheads, primarily from *enclave context switches*, upon every CPU transition between non-enclave and enclave code. The Intel SGX SDK [6] provides specialised function call mechanisms (*i.e.*, `ecalls` and `ocalls`), to enter, and respectively exit, an enclave. Enclave switches cost up to 14,000 CPU cycles [32], [33], on average 56× more expensive compared to regular system calls on similar Intel CPUs (*i.e.*, 250 cycles [21]). This represents a serious performance bottleneck for applications which perform

many enclave context switches, *e.g.*, to perform system calls via `ocalls` [33].

Techniques exist [4], [28], [33] which circumvent expensive enclave switches by leveraging *worker threads* in and out of the enclave. *Client threads*, *i.e.*, inside or outside of the enclave, send their requests to the opposite side via shared memory. Worker threads handle such requests and send the appropriate responses once completed. The Intel SGX SDK implements this technique via the *switchless call library* [8]. While this approach improves the performance of enclave context switches, our practical experience revealed the following problems. First, Intel SGX switchless calls **must be manually configured** at build time and misconfigurations can lead to performance degradations and waste of CPU resources [8]. Second, manually configuring `ecalls` or `ocalls` as switchless at build time is not ideal: developers rarely know the frequency nor the duration of the calls, which are typically application and workload specific.

To mitigate these problems, we propose ZC-SWITCHLESS (*zero-config* switchless), a system to **dynamically select switchless routines** at run time and **configure the most appropriate number of worker threads on the fly**. This approach prevents static configuration of switchless routines and worker threads, and obviates the performance penalty of misconfigured switchless systems, while minimizing CPU waste.

ZC-SWITCHLESS leverages an application level scheduler that monitors runtime performance metrics (*i.e.*, number of wasted cycles or fallback calls to regular non-switchless routines, *etc.*); these are used to dynamically configure an optimal number of worker threads to fit the current workload while minimising CPU waste.

We further analysed the Intel SGX SDK implementation, and derived practical configuration tips for Intel SGX switchless calls. In addition, we tracked performance issues with Intel’s SDK `memcpy` implementation, and propose an optimised version which removes a major source of performance overhead for both switchless and regular SGX transition routines.

In summary, our **contributions** are: (1) The design and the implementation of ZC-SWITCHLESS, a configless and efficient system to dynamically select and configure switchless calls, to be released as open-source. (2) An optimised implementation for the Intel SGX SDK’s `memcpy`, also to be open-sourced. (3) An extensive experimental evaluation demonstrating the effectiveness of our approach via micro- and macro-benchmarks.

II. BACKGROUND ON INTEL SGX

Intel SGX extends Intel’s ISA to enable secure memory regions called *enclaves*. The CPU reserves an encrypted portion of DRAM, called the *enclave page cache* (EPC), to store enclave code and data at runtime. EPC pages are only decrypted inside the CPU by the *memory encryption engine* (MEE) once loaded in a CPU cache line.

Enclaves operate only in user mode and thus cannot issue system calls (syscalls) [9] directly. The `ecalls` and `ocalls` are specialised function calls to enter (using the `EENTER` CPU instruction) and exit (using the `EEXIT` CPU instruction) the enclave, respectively. The Intel SGX SDK provides a secure version of the C standard library, the *trusted libc* (`tlIBC`), for in-enclave applications. The `tlIBC` only relies on trusted functions, and removes instructions forbidden by Intel SGX [8].

Enclave applications can be deployed following two approaches: (1) running the entire application in the enclave (e.g., SCONE [4], Graphene-SGX [5], Occlum [27], SGX-LKL [23]), or (2) splitting the application into a trusted and untrusted part which respectively execute inside and outside of the enclave (e.g., Glamdring [15], Civet [30], Montsalvat [34]).

For both cases, unsupported routines not implemented by the `tlIBC` must be relayed to the untrusted part via `ocalls`, to be executed in non-enclave mode.

Both `ecalls` and `ocalls` perform expensive CPU context switches, i.e., switching from enclave to non-enclave mode, and vice-versa. The overhead of enclave transitions is mainly due to the CPU flushing its caches as well as all TLB entries containing enclave addresses, so as to preserve confidentiality [9]. More precisely: an `ocall` = `EEXIT` + untrusted host processing + `EENTER`. `EEXIT` flushes the CPU caches and possibly invalidates branch predictors and TLBs. Similarly, the `EENTER` instruction performs many checks and requires hardware-internal synchronization of CPU cores [26].

In the following, we focus on `ocalls` as they are usually the principal source of overhead due to enclave context switches, based on our experience. However, the techniques we propose in this paper can equally be used for `ecalls`.

Switchless `ocalls`. The Intel SGX SDK provides two variants of `ocalls`. First, *regular `ocalls`* perform costly context switches (up to 14,000 cycles [32]) from enclave mode to non-enclave mode. Second, *switchless `ocalls`* [8] provide a mechanism to do cross-enclave calls without performing an enclave context switch. In a nutshell (i.e., Fig. 1), *worker threads* outside the enclave perform unsupported functionality

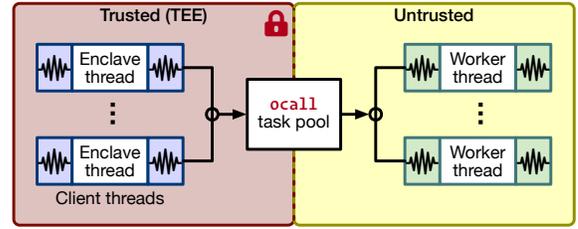


Fig. 1: Intel SGX switchless `ocall` architecture.

(e.g., syscalls) on behalf of in-enclave client threads. The latter send `ocall` requests to a task pool in untrusted memory. The worker threads outside the enclave wait for pending tasks in the task pool, and execute them until the task pool is empty [8]. These operations are done without performing any enclave transition.

III. LIMITATIONS OF INTEL SGX SWITCHLESS CALLS

In the following, we identify three main problems and limitations with the Intel switchless calls implementation: (i) switchless call selection (§III-A), (ii) worker thread pool sizing (§III-B), and (iii) Intel SDK parameterisation (§III-C).

Setup. We use a 4-core (8 hyper-threads) Intel Xeon CPU E3-1275 v6 clocked at 3.8 GHz, 8 MB L3 cache, 16 GB of RAM, with support for Intel SGX v1. We deploy Linux Ubuntu 18.04.5 LTS with kernel 4.15.0-151 and Intel SGX SDK v2.14. We report the median over 10 executions.

A. Switchless calls selection

At build-time, developers must specify the routines (i.e., `ecalls` or `ocalls`) to be handled switchlessly at run time. The Intel SGX reference [8] suggests to configure a routine as switchless if it has *short* duration and is *frequently called*. However, these details are hardly available to the developer at build time. Auto-tuning tools [18] cannot spot potential switchless routines by relying solely on duration and frequency, as it would require a full code path exploration, a costly operation for large systems. Further, the execution frequency of a specific application routine is workload specific and hard to configure at build time.

Improperly selected switchless routines degrade the performance of SGX applications. We show this behaviour with a synthetic benchmark. It executes n `ocalls` to 2 functions as follows: α calls to function f which is known to benefit from switchless calls (as shown in [28]), while β calls to g which should run as a regular `ocall`. If $\alpha = \beta$, the sum of the execution times of calls to f is negligible compared to the same sum for calls to g . Hence, to better highlight the performance gains when executing f switchlessly, we set $\alpha = 3\beta$ and $n = \alpha + \beta$. In this test, f is an empty function (i.e., `void f(void){}`). On the other hand, g routine executes `asm("pause")` in a loop, i.e., a busy-wait loop.

We evaluate five different configurations: C1-C5. In C1, all f functions run switchlessly, while g functions run as regular `ocalls`. We expect C1 to perform best. In C2, only

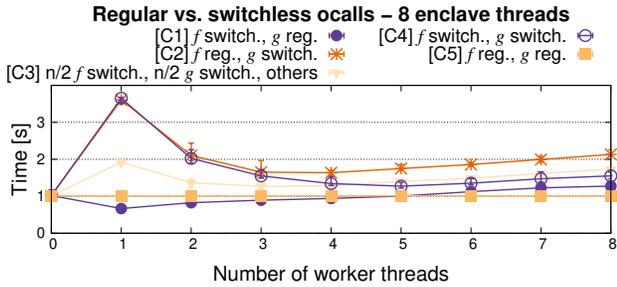


Fig. 2: Runtime for 75,000 switchless `ocalls` to f and 25,000 regular `ocalls` to g

the g functions run switchlessly, and we expect C2 to be the worst. In the C3 case, $\frac{\alpha}{2} f$ and $\frac{\beta}{2} g$ functions run switchlessly while the other f and g functions run as regular `ocalls`. Finally, in C4 and C5 all functions run switchlessly or regularly, respectively. We observe the following results when executing 100'000 `ocalls` (*i.e.*, switchless and regular `ocalls` combined). C1 is the fastest configuration (0.9s), $\approx 1.8\times$ faster than C2, indeed the worst (1.6s). C3 and C4 complete in 1.3s ($1.44\times$ slower than C1). Finally, C5 completes in 1s.

Take-away 1: An improper selection of switchless coroutines degrades the performance of SGX applications.

B. Worker thread pool sizing

Developers must specify the total number of worker threads allowed to the SGX switchless call mechanism at build time. However, an overestimation of worker threads for `ocalls` can lead to a waste of CPU resources due to busy-waiting of worker threads awaiting switchless requests (see [8], page 71). The latter will limit the number of applications that can be co-located on the same server or interfere with application threads which will be deprived of CPU resources. Similarly, an underestimation can lead to poor application performance as more `ocalls` will perform costly enclave switches.

Using the same synthetic benchmark from §III-A, we validate these effects for a varying number of worker threads (see Fig. 2). The optimal number of workers strictly depends on the specific configuration. At its best (C1), the fewer the workers, the better the performance: this is expected, as there are as many in-enclave threads as hardware cores. In the other cases, we observe better results when executing functions as regular `ocalls`, where the best overall result is using 4 worker threads.

Fig. 3 presents the execution time of the application depending on both the number of worker threads (from 1 to 5) and the duration of g (from 0 to 500 `asm{"pause"}` instructions, each spending 140 CPU cycles). We report 4 configurations (we omit C3 for the sake of clarity and because it obtains results between the worst of C1 and C2). We can make the following observations. C5 (*i.e.*, both f and g as regular `ocalls`) performs worst on average for the shortest g function (*i.e.*, 0 pauses), but it is best in several cases for

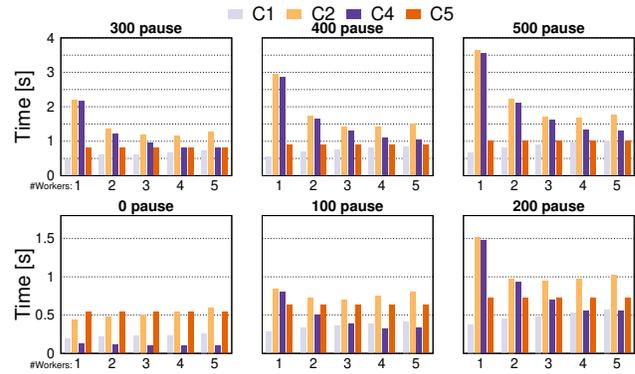


Fig. 3: Runtime for 100,000 `ocalls` with 8 in-enclave threads for different durations of g function.

longer g functions, regardless of the number of workers. C1 (f switchless, g regular) is the best when g is longer than 200 pauses. Executing all functions switchlessly (C4) is good for short g functions, scaling with the available worker threads.

Take-away 2: Switchless calls perform best when the calls are short, relative to the cost of an enclave transition.

C. Intel SDK parameterisation

If a switchless task pool is full or all worker threads are busy, a switchless call falls back to a regular `ecall` or `ocall`. The Intel SGX SDK defines the variable `retries_before_fallback` (`rbf`) for the number of retries client threads perform in a busy-wait loop, waiting for a worker thread to start executing a switchless call request, before falling back to a regular `ecall/ocall` [8]. Similarly, the SDK defines `retries_before_sleep` (`rbs`), for the number of `asm{"pause"}` done by a worker while waiting for a switchless request, before going to sleep. The SDK's default values for both `rbf` and `rbs` are set at 20,000 retries.

However the value of `rbf` especially is abnormal in both a theoretical (as we explain next) and practical sense: between successive retries, a caller thread executes an `asm{"pause"}` instruction, which has an estimated latency up to 140 cycles on Skylake-based microarchitectures (where SGX extensions were first introduced, see <https://intel.ly/3hTVEMG>, page 58). A caller thread can hence wait more than 2.8M cycles before its call is handled by a worker thread. This is about $200\times$ more costly relative to a regular `ocall` transition ($\approx 14,000$ cycles), and defeats the purpose of using switchless calls, *i.e.*, to avoid the expensive transition. Similarly for `rbs`, a worker thread will wait for 2.8M cycles before going to sleep.

While developers can easily tune the `rbf` and `rbs` values at build time, the Intel SDK lacks proper guidance.

Take-away 3: The proper configuration of the Intel SGX switchless-related parameters remains hard and misconfigurations lead to poor performance.

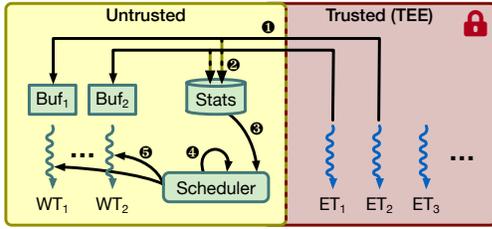


Fig. 4: ZC-SWITCHLESS general overview.

IV. ZC-SWITCHLESS

The main goal of ZC-SWITCHLESS is to provide a resource-efficient implementation for SGX switchless calls.

We first present its internal scheduler in §IV-A-IV-C. In addition, we detail a more efficient implementation of `memcpy` (§IV-F), used for intra-enclave data copying, as well as data exchange between the enclave and the outside world.

Fig. 4 shows an overview of ZC-SWITCHLESS. In a nutshell, we consider *any function* (Fig. 4-①) as a potential candidate to run as switchless, thus avoiding the need for manual selection by developers at build time. Our design allows the number of worker threads to be tuned dynamically, to minimise CPU waste while improving performance via the switchless call technique. We do so by having the scheduler implement a feedback loop (Fig. 4-②) to periodically collect `ocall` statistics (Fig. 4-③), determine the optimal number of worker threads (Fig. 4-④), and finally apply its decision (Figure 4-⑤). Inside the enclave, a call is executed in a switchless manner if the caller finds at least one idle worker thread. The remainder details further these aspects. Unless indicated otherwise, the term *scheduler* refers to ZC-SWITCHLESS’s scheduler.

A. ZC-SWITCHLESS’s scheduler

The main objective of the scheduler is to minimise wasted CPU cycles. We define a *wasted CPU cycle* as one spent by a CPU core doing something that does not make the application (*i.e.*, caller thread) move forward in its execution [16].

In the case of Intel SGX, we identify two potential sources of wasted CPU cycles: (1) transitions between enclave and non-enclave mode in the case of regular `ocalls`, and (2) busy-waiting in the case of switchless `ocalls`. The overhead of regular `ocalls` has been evaluated extensively in past research [33], and it varies also according to the specific CPU and micro-code version. We evaluated this overhead to be $\sim 13,500$ CPU cycles for our experimental setup (see §III).

Each active worker can be at any point in time in one of two states: either the worker is handling an `ocall`, in which case the enclave thread (which made the `ocall`) is busy-waiting, or the worker is busy-waiting for incoming `ocall` requests. Therefore, for every active worker thread, there is always exactly one thread busy-waiting. The extra cost of having M worker threads is thus M multiplied by the number of cycles during which they have been active.



Fig. 5: ZC-SWITCHLESS scheduler phases.

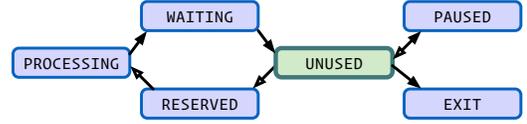


Fig. 6: Worker thread state transitions.

The scheduler periodically computes the number of worker threads that minimises the number of wasted CPU cycles. Throughout its lifetime, the scheduler switches between two phases (see Fig. 5): a *scheduling phase* that lasts a scheduler quantum, Q , during which it sets an optimal number of switchless workers, and a *configuration phase*, during which it calculates the optimal number of switchless workers for the next scheduling phase.

We denote by T_{es} the duration of an enclave switch, F the number of calls not being handled switchlessly (*i.e.*, fallback), N the number of cores on the machine and M the number of worker threads set during ZC-SWITCHLESS’s scheduler quantum (Q , set empirically to 10 ms).

The number of wasted cycles during T cycles is: $U = F \cdot T_{es} + M \cdot T$. At every quantum (*i.e.*, during a *configuration phase*), the scheduler thread estimates the optimal number of workers for the next quantum (*i.e.*, *scheduling phase*).

To estimate the number of workers during a *configuration phase*, the scheduler sleeps for $\frac{N}{2} + 1$ micro-quanta, of length $\mu \cdot Q$ each, with a different number of workers i each time, $0 \leq i \leq \frac{N}{2}$ (*i.e.*, $\frac{N}{2} + 1$ possible values). The constant μ is a small time period, so the configuration phase can be quick, but still long enough to capture the needs (in terms of CPU resources) of the application at a given time. We empirically set $\mu = \frac{1}{100}$. During every micro-quantum, the number of non-switchless calls is recorded so that the scheduler can compute $U_i = F_i \cdot T_{es} + i \cdot \mu \cdot Q \cdot CPU_FREQ$ once awake. Finally, the scheduler keeps M' workers for the next *scheduling phase*, where M' is such that $U_{M'} = \min_i U_i$.

To deactivate a worker thread, the scheduler sets a value in the worker’s buffer (see §IV-B). The worker’s loop function will eventually check this value and, if it is set and no caller thread has reserved (or is using) the worker, the worker will pause. To re-activate a paused worker, the scheduler sends a signal to wake up the corresponding worker thread.

B. Worker thread state machine

We associate to each worker a `buffer` structure that consists of 4 main fields: an untrusted memory pool (preallocated) used by callers to allocate switchless requests, a field to hold the most recent switchless request, a status field to track the worker status, and a field used to communicate with the scheduler. Fig. 6 summarises the `status` transitions of a worker thread.

A worker is initially in the `UNUSED` state. When a caller needs to make a switchless call, it finds an `UNUSED` worker and switches the worker’s state to `RESERVED`. We rely on GCC atomic built-in operations [10] for thread synchronisation. The caller allocates a switchless request structure from the corresponding memory pool. A switchless request comprises: an identifier of the function to be called, the function arguments (if present), and the return result (if present).

The caller copies its request to the worker’s buffer and changes the worker’s state from `RESERVED` to `PROCESSING`. At this point, the worker reads the request and calls the desired function with the corresponding arguments. Once the function call completes, the worker updates the request with the returned results (if present) and switches from the `PROCESSING` to the `WAITING` state.

Finally, the caller copies the returned results into enclave memory and changes the worker’s state to `UNUSED`.

The memory pools of worker buffers are freed and re-allocated when full via an `ocall`. Using preallocated memory pools prevents callers from performing `ocalls` to allocate untrusted memory for each switchless request, which will defeat the purpose of using a switchless system.

Upon program termination, the scheduler sets a value in workers’ buffers so the workers can switch to the `EXIT` state. At this stage, the workers perform final cleanup operations (e.g., freeing memory) and then terminate.

C. Switchless call selection

In `ZC`, any routine can be run as switchless if the corresponding enclave caller thread finds an available/unused worker thread. Otherwise, the call immediately falls back to a regular `ocall` without any busy waiting.

D. Integrating ZC switchless with other TEE implementations

Other popular TEE implementations operate following a very similar architecture to Intel SGX. For example in ARM TrustZone (Armv8-M architecture) [22], the application is divided into two parts: a secure world (i.e., enclave) with very limited system functionality, and a normal world (untrusted world) with a richer system API. Similar to Intel SGX, CPU thread transitions between the secure and normal worlds require extra security checks to guarantee data confidentiality and integrity. So conceptually, the design proposed in `ZC` can be applied here. This will, however, require some modifications at the implementation level.

E. Security analysis of ZC-SWITCHLESS

The security analysis of `ZC-SWITCHLESS` is similar to that presented in [33]. All switchless call designs (i.e., Intel switchless, `ZC-SWITCHLESS`, hotcalls [33]) are based on threads in and out of the enclave communicating via plaintext shared memory. Thus, the switchless design proposed by `ZC-SWITCHLESS` is no less secure than that proposed by the Intel SGX switchless library.

Impact of ZC scheduler on security. The `ZC-SWITCHLESS` scheduler is located in the untrusted runtime,

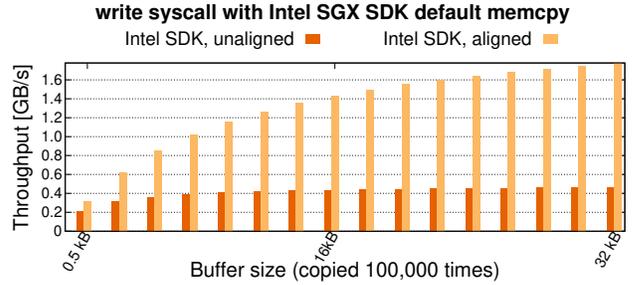


Fig. 7: Throughput for `ocalls` of the `write` system call to `/dev/null` (100,000 operations, average over 10 executions) for aligned and unaligned buffers.

which leaves it vulnerable to malicious tampering. However, since the scheduler only decides how many worker threads should be used and when, the worst case scenario here will be a DoS, e.g., by killing worker threads. The enclave’s confidentiality or integrity, however, cannot be compromised by tampering with the scheduler.

F. Trusted libc’s memcpy optimisation

For security reasons, the Intel SDK provides its own implementation of a subset of the functions from the `libc`, i.e., the `tllibc`. The `tllibc` re-implements most of the functions from the `libc` that do not require system calls, e.g., `memset`, `memcpy`, `snprintf`, etc. Because enclave code cannot be linked to dynamic libraries, these re-implementations are statically linked to the enclave application at build time.

We focus on the Intel SDK `tllibc` version of `memcpy` [2], as it is heavily used to pass `ocall` arguments from trusted memory to untrusted memory and back for the results [14]. Our tests highlighted huge performance gaps when using aligned buffers (i.e., when the `src` and `dest` arguments are congruent modulo 8) and unaligned buffers.

For instance, Fig. 7 presents the throughput when issuing 100,000 `write` system calls, which requires an `ocall` and involves `memcpy` calls, with varying lengths of aligned and unaligned buffers ranging from 512 B to 32 kB. We observe that the execution time for unaligned buffers is consistently higher than for aligned buffers. Moreover, when using unaligned buffers, we observe poor scalability trends for `write` when increasing the buffer sizes, basically plateauing at about 0.4 GB/s.

By analysing Intel’s original implementation of `tllibc`’s `memcpy`, we observed that it performs a *software word-by-word copy for aligned buffers* but a *byte-by-byte copy for unaligned buffers*. We provide a revised and more efficient implementation leveraging the hardware copy instruction `rep movsb`, as also advised by Intel’s optimization manual [1]. Listing 1 sketches our optimised approach, which we evaluate in depth in §V-D.

V. EVALUATION

Our evaluation answers the following questions:

```

1 void *memcpy(void *dst0, const void *src0, size_t length) {
2     ...
3     /* Copy forward. */
4     __asm__ volatile(
5         "rep movsb"
6         : "=D"(dst0), "=S"(src0), "=c"(length)
7         : "0"(dst0), "1"(src0), "2"(length)
8         : "memory");
9 done:
10    return (dst0);
11 }

```

Listing 1: ZC-SWITCHLESS optimised memcpy.

- Q1) How does ZC-SWITCHLESS impact application performance for (1) static workloads (§V-A1), and (2) dynamic workloads (§V-C1)?
- Q2) What is the effect of misconfigurations of Intel switchless on application performance? (§V-A1) and (§V-C1)?
- Q3) What is the effect of ZC-SWITCHLESS on CPU utilisation for static (§V-A2) and dynamic (§V-C2) workloads?
- Q4) What is the performance gain of our improved memcpy implementation (§V-D) ?

Experimental setup. All experiments use a server equipped with a 4-core Intel Xeon CPU E3-1275 v6 clocked at 3.8 GHz with hyperthreading enabled. The CPU supports Intel SGX, and ships with 32 KB L1i and L1d caches, 256 KB L2 cache and 8 MB L3 cache. The server has 16 GB of memory and runs Ubuntu 18.04 LTS with Linux kernel version 4.15.0-151. We run the Intel SGX platform software, SDK, and driver version v2.14. All our enclaves have maximum heap sizes of 1 GB. The EPC size is 128 MB (93.5 MB usable by enclaves). We use both static and dynamic benchmarks.

Our static benchmarks are based on `kissdb` [12] and an Intel SGX port of OpenSSL [7]: `kissdb` is a simple key/value store implemented in plain C without any external dependencies, while OpenSSL [24] is an open-source software library for general-purpose cryptography and secure communication.

For dynamic benchmarks, we use `lmbench` [20], a suite of simple, portable, ANSI/C microbenchmarks for UNIX/POSIX.

For all benchmarks, we set the initial number of worker threads to $\frac{\#logical_cpus}{2}$ for ZC-SWITCHLESS. This number is 4 for the SGX server used. For Intel switchless experiments, we maintain the default `rbf` and `rbs` values (*i.e.*, 20,000).

A. Static benchmark: `kissdb`

We issue a varying number of key/value pair writes to `kissdb`, and we evaluate and compare the performance and CPU utilisation of ZC-SWITCHLESS with Intel switchless.

We ran our benchmark in 3 modes: without using switchless calls (`no_sl`), using Intel switchless calls, and using ZC-SWITCHLESS (`zc` for short). For Intel switchless, we consider two values for the number of switchless worker threads: 2 and 4. In `kissdb`, we know empirically that the 3 most frequent `ocalls` in the benchmarks are: `fseeko`, `fwrite`, and `fread`. Therefore, for `kissdb` we benchmark Intel’s switchless in 10 (2×5) different configurations: only `fseeko` as switchless (`i-fseeko-x`, x being 2 or 4, the number of

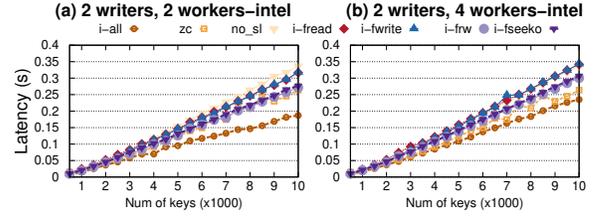


Fig. 8: `kissdb`: average latency of key/value SET commands.

Intel switchless worker threads), only `fwrite` as switchless (`i-fwrite-x`), only `fread` as switchless (`i-fread-x`), both `fread` and `fwrite` as switchless (`i-frw-x`), and all the 3 `ocalls` as switchless (`i-all-x`). Note that these ten configurations correspond to possible configurations an SGX developer could have set up for `kissdb`. We report the averages over 5 runs.

1) ZC-SWITCHLESS vs. Intel switchless: Answer to Q1 & Q2. Fig. 8 (a) and (b) show the average latencies for setting a varying number of 8-byte keys and 8-byte values in `kissdb`, with respectively 2 and 4 switchless worker threads configured for Intel switchless.

Observation. Here, `zc` is 1.22× faster when compared to a system without switchless calls, and respectively 1.19×, 1.13×, 1.13×, 1.05×, and 1.02× faster when compared to `i-fread-2`, `i-fwrite-2`, `i-fseeko-2`, and `i-frw-2`.

However, `zc` is about 1.33× slower for `i-all-2`. We note how `zc` is (on average) 1.26× faster than `i-fread-4`, 1.22× faster than `i-fwrite-4`, 1.13× faster than `i-fseeko-4`, 1.10× than `i-frw-4`. However, it is 1.16× slower than `i-all-4`.

Discussion. In `kissdb`, `fseeko` is the most frequent `ocall`, invoked almost twice more often than `fread` and `fwrite`. Further, `fseeko` is much shorter in duration relative to `fread` and `fwrite`, which explains the better performance of `i-fseeko` when compared to `i-fread` and `i-fwrite` for both 2 and 4 switchless worker threads. `i-fwrite` configurations show the poorest performance for Intel’s switchless in all cases.

However, when both `fread` and `fwrite` are configured as switchless (`i-frw`), we see an improvement in performance relative to `i-fwrite` and `i-fread`, almost equal to `i-fseeko` performance. Here the combined sum of `fread` and `fwrite` calls surpasses the number of `fseeko` invocations, which leads to a more significant number of switchless calls in `i-frw`, thus leading to similar performance as `fseeko`.

This configless strategy of `zc` outperforms statically misconfigured systems like `i-fread` and `i-fwrite` (for both 2 and 4 Intel switchless workers), shows similar performance as `i-fseeko-2`, and is faster than `i-fseeko-4`. The observed spikes (*e.g.*, 7,500 and 8,500 keys) in `zc` are due to `ocall` operations when reallocating full memory pools for `zc` buffers (see §IV-B).

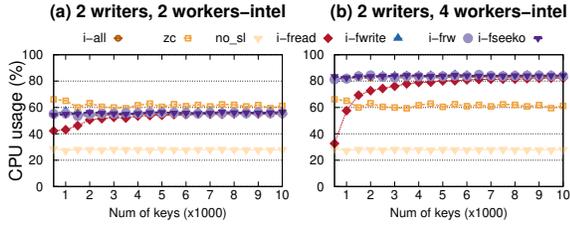


Fig. 9: Average %CPU usage for key/value pair SET ops in kissdb

Take-away 4: ZC-SWITCHLESS achieves better performance relative to non-switchless systems, and outperforms misconfigured Intel switchless systems.

In *i-all*, Intel’s switchless outperforms *zc* because it maintains a constant number of switchless workers (2 or 4), whereas *zc* uses few worker threads at various points during the application’s lifetime.

Take-away 5: A well configured Intel switchless system outperforms *zc*, but *zc* obviates the performance penalty observed from the misconfigured switchless systems.

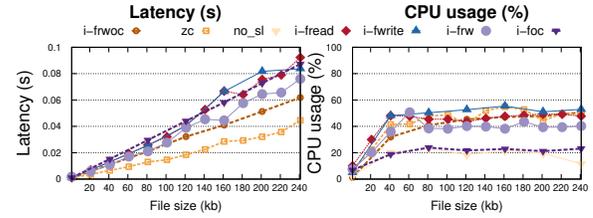
2) ZC-SWITCHLESS vs. Intel switchless: CPU usage:

Answer to Q3. We now evaluate the CPU utilisation of ZC-SWITCHLESS and Intel switchless when running the same experiments as in §V-A1. We measured the overall CPU utilisation for the given systems from the kernel’s `/proc/stat`. The percentage CPU utilisation is calculated by: $\%cpu_used = \frac{(user+nice+system)}{(user+nice+system+idle)} * 100$ where: *user* is the time spent for normal processes executing in user mode, *nice* is the time spent for processes executing with “nice” priority in user mode, *system* is the time spent for processes executing in kernel mode, and *idle* is time spent by the CPU executing the *system idle process*.

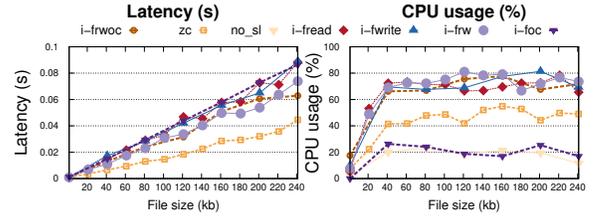
Fig. 9 shows the average CPU usage for setting a varying number of 8-byte keys and 8-byte values in *kissdb*, with respectively 2 and 4 switchless worker threads configured for Intel switchless.

Observation. The experimental results show that *zc* maintains approximately 60% CPU usage throughout the benchmark’s lifetime. For 2 Intel switchless workers, all Intel switchless configurations stabilise at about 55% CPU usage, while for 4 Intel switchless workers, the Intel switchless configurations stabilise at about 80% CPU usage. All switchless systems have visibly higher CPU usage when compared to the system without switchless calls enabled (*no_sl*).

Discussion. Intel’s switchless mechanism maintains a constant number of worker threads (2 or 4) throughout the application’s lifetime, while *zc*’s scheduler increases or decreases the number of worker threads (to a maximum of 2 or 4) with respect to the workload. This explains the overall lower CPU usage of *zc* relative to Intel switchless.



(a) 2 Intel workers.



(b) 4 Intel workers.

Fig. 10: Latency and CPU usage for OpenSSL.

Take-away 6: *zc* outperforms misconfigured systems (e.g., *i-fread*, *i-fwrite*, *i-frw*) while minimizing CPU usage.

B. Static benchmark: OpenSSL file encryption/decryption

This benchmark consists of two enclave threads encrypting and decrypting data read from files. The first thread reads chunks of plaintext from a file, encrypts these in the enclave, and writes the corresponding ciphertext to another file, while the second thread reads ciphertext from a different file, and decrypts it inside the enclave. All cryptographic operations are done with the AES-256-CBC [25] algorithm.

Similarly, we ran this benchmark in the 3 modes described above: *no_sl*, using Intel switchless calls (2 and 4 workers), and *zc*.

In the OpenSSL benchmark, we know empirically that 4 *ocalls* are called most frequently: *fread*, *fwrite*, *fopen*, and *fclose*. We consider 10 possible configurations (2×5) for Intel’s switchless routines: only *fread* as switchless (*i-fr-x*, *x* being 2 or 4, the number of Intel switchless worker threads), only *fwrite* as switchless (*i-fw-x*), both *fread* and *fwrite* (*i-frw-x*), both *fopen* and *fclose* (*i-foc-x*), and all 4 *ocalls* as switchless (*i-frwoc-x*).

Fig. 10a and Fig. 10b show the latency and CPU usage for these configurations. We highlight and discuss essential observations.

Observation and discussion. In this OpenSSL benchmark, *fread* and *fwrite* are respectively called $\approx 2700 \times$ and $\approx 1400 \times$ more frequently as compared to both *fopen* and *fclose*. This explains why Intel’s latency is poor (close to *no_sl*) with *i-foc*, as more *ocalls* perform context switches, and much better (w.r.t *no_sl*) with *i-frw*, where a larger number of *ocalls* are performed switchlessly. Intel performs best with *i-frwoc*, when the four *ocalls* are all

configured as switchless. However, we observe that `zc` is about $1.62\times$ and $1.82\times$ faster than Intel’s best configuration `i-frwoc`, for 2 and 4 Intel workers respectively. This is explained by the fact that the `fread` and `fwrite` calls here are much longer (about $6\times$ longer as compared to the previous `kissdb` benchmark). This accentuates the bad effect of the poor default `rbf` value in Intel’s switchless, as enclave threads do longer pauses waiting for a switchless worker thread to become available. This is absent in `zc`, where caller threads immediately fall back.

Regarding CPU usage, `zc`’s scheduler set the number of worker threads to 0, 1, 2, 3, 4 for respectively 9.4%, 4.6%, 84.4%, 1.6%, and 0% of the program’s lifetime. This explains the similar CPU usage in `zc` and Intel 2 workers (except for `i-foc`), while with 4 Intel workers, CPU usage for Intel’s best config is about $1.62\times$ larger than `zc`’s, despite the latter performing better.

Take-away 7: `zc` outperforms all Intel configurations when `ocalls` are long; this is due to the poor default `rbf` value in Intel’s switchless library.

C. Dynamic benchmark: `lmbench`

Our dynamic benchmark is based on the `read` and `write` system call benchmarks of `lmbench`. The `read` benchmark iteratively reads one word from the `/dev/zero` device [20], while the `write` benchmark iteratively writes one word to the `/dev/null` device. We devised a dynamic workload approach which consists of periodically (every $\tau = 0.5s$) issuing a varied number of read and write operations to `lmbench` using two in-enclave caller threads (1 reader + 1 writer) over a period of 60s. These operations trigger `ocalls`, and `zc` scheduler adapts the number of worker threads accordingly.

The total run time of the dynamic benchmark is divided into 3 distinct phases, each lasting 20s: (1) *increasing operation-frequency*: the number of operations is doubled periodically, (2) *constant operation-frequency*: the number of operations remains at a constant value (the peak value from phase-1), and (3) *decreasing operation-frequency*, where the number of operations is periodically decreased (reduced by half every τ). We measure the read/write throughputs and CPU usage at different points during the benchmark’s lifetime.

Similarly, we ran our benchmark in 3 modes: without using switchless calls (`no_sl`), using Intel switchless calls, and using ZC-SWITCHLESS (`zc`). For Intel switchless, we consider two values for the number of switchless worker threads: 2 and 4. For `lmbench` syscall benchmark, we know empirically that the `read` and `write` syscalls are the most frequent `ocalls`. So we configure Intel’s switchless in six (3×2) different configurations: only `write` as switchless (`i-write-x`), only `read` as switchless (`i-read-x`), both `read` and `write` `ocalls` as switchless (`i-all-x`). Similarly, these six configurations correspond to possible configurations an SGX developer could have set for their program. We show the throughputs and CPU usages as observed from both the reader and writer threads. We highlight essential observations, and analyze them in our discussions.

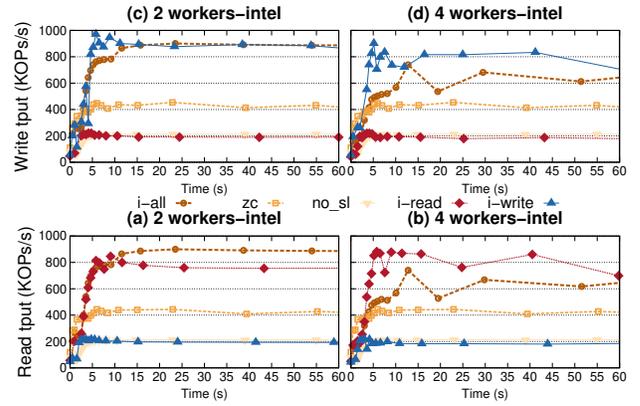


Fig. 11: Read (a/b), write (c/d) tput. for `zc` vs. 2/4 Intel switchless worker threads.

1) *Dynamic benchmark: ZC-SWITCHLESS vs. Intel switchless: Answer to Q1 & Q2.* Fig. 11 shows the operation throughputs as observed by the reader (bottom) and writer (top) threads respectively during the lifetime of the dynamic benchmark.

Observation. The experimental results show that on average, `zc` is about $2.3\times$ faster when compared to both `reader-i-write-2` and `reader-i-write-4`, and $2.1\times$ and $2.5\times$ faster when compared to `writer-i-read-2` and `writer-i-read-4` respectively. However, `zc` is about $1.6\times$ and $1.1\times$ slower when compared to properly configured Intel switchless configurations `i-all-2` (reader,writer) and `i-all-4` (reader, writer) respectively.

Discussion. From the perspective of the reader thread, `i-write` is a misconfiguration as the `read` calls will never be switchless, and similarly for the writer thread, `i-read` is a misconfiguration as the `write` calls will never be switchless. This explains the relatively lower throughputs of these configurations when compared to `zc` and the other switchless configurations. However, `zc` has a lower throughput when compared to the better configurations: `reader-(i-all,i-read)` and `writer-(i-all,i-write)`.

2) *ZC-SWITCHLESS CPU utilisation vs. Intel switchless: Answer to Q3.* We compute the CPU usage as explained previously. Fig. 12 shows the average CPU usage as observed by the reader (top) and writer (bottom) threads respectively at the different points during the lifetime of the dynamic benchmark.

Observation. Similarly to the throughputs, the CPU usage for the studied configurations increases with time and plateaus at a certain point. The experimental results show that on average, `zc` CPU usage is about $1.8\times$ and $1.6\times$ more when compared to `reader-i-write-2` and `writer-i-read-2` respectively, but almost equal CPU usage on average when compared to `reader-i-write-4`, `writer-i-read-4`, and `reader/writer-i-all-2` respectively. However, `reader/writer-i-all-4` use about $1.3\times$ more CPU when compared to `zc`.

Discussion. Similarly to the poor throughputs, we can eas-

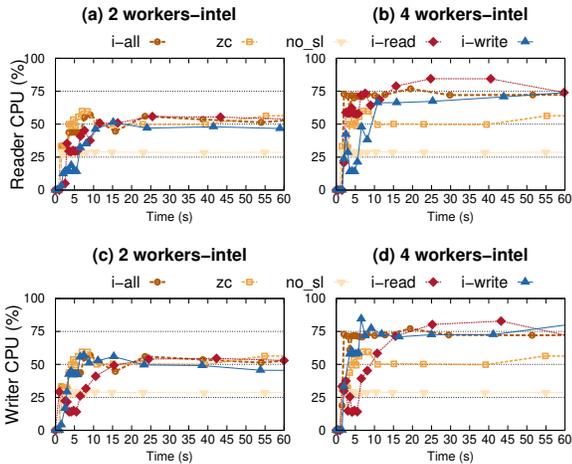


Fig. 12: Read(top)/Write(bottom) CPU usage % for `zc` vs. 2/4 Intel switchless worker threads.

ily highlight CPU waste for the misconfigurations: reader-`i-write-4` and writer-`i-read-4`, as they have similar CPU usages when compared to `zc` on average but much poorer performance with regards to the corresponding throughputs. `ZC-SWITCHLESS` prevents this misconfiguration problem. For the better configurations: reader-`(i-read,i-all)`, and writer-`(i-write,i-all)`, Intel performs better than `zc` but this usually comes at a higher CPU cost (especially for 4 Intel workers), which is explained by the fact that Intel’s switchless mechanism maintains the maximum number of workers when there are pending switchless requests.

Take-away 8: *Poorly configured Intel switchless systems lead to a waste of CPU resources. `ZC-SWITCHLESS` obviates the performance penalty from misconfigured Intel switchless systems, while minimising CPU waste.*

D. Performance of improved `memcpy`

Answer to Q4. To evaluate the performance of our improved `memcpy` implementation, we ran a benchmark similar to that in §IV-F, which issues 100,000 `write` system calls from within the enclave, with various sizes of aligned and unaligned buffers ranging from 512 B to 32 kB. We ran this benchmark in two modes: using the default `memcpy` implementation of the SDK (`vanilla-memcpy`) and using our improved `memcpy` implementation (`zc-memcpy`).

As shown in Fig. 13, our revised `memcpy` implementation achieves a speedup, in the case of larger buffers, of up to $3.6\times$ for aligned buffers and $15.1\times$ for unaligned buffers. Noteworthy, these speedups will benefit both regular and switchless `ocalls`, as well as any application using the Intel SDK’s `memcpy` implementation inside enclaves.

Impact on inter-enclave communication. Recent work [14] presents a quantitative study on the performance of real-world serverless applications protected in SGX enclaves. A performance test of `zc-memcpy` in this work showed a 7% – 15% speedup for inter-enclave SSL transfers in the

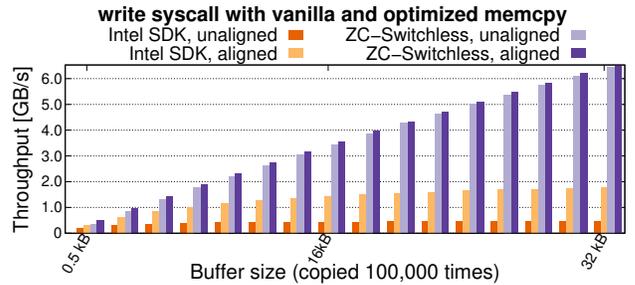


Fig. 13: Throughput for `ocalls` of the `write` system call to `/dev/null` (100,000 operations, average over 10 executions) for aligned and unaligned buffers.

context of their benchmarks, which confirms the efficiency of `zc-memcpy` for copy-intensive operations.

VI. RELATED WORK

We classify related work into three categories, as detailed next.

(1) **SGX benchmarking and auto-tuning tools.** In [18], authors use stochastic optimisation to enhance the performance of applications hardened with Intel SGX. [17] proposes a framework for benchmarking SGX-enabled CPUs, micro-code revisions, SDK versions, and extensions to mitigate enclave side-channel attacks. These tools do not provide dynamic configuration of the switchless mechanisms.

(2) **SGX performance improvement.** Weichbrodt et al. [32] propose a collection of tools for high-level dynamic performance analysis of SGX enclave applications, as well as recommendations on how to improve enclave code and runtime performance, e.g., by batching calls, or moving function implementations in/out of the enclave to minimise enclave transitions. *Intel VTune Profiler* [31] permits to profile enclave applications to locate performance bottlenecks, while Dingding et al. [13] provide a framework to improve enclave creation and destruction performance. `ZC-SWITCHLESS` focuses on improving enclave performance efficiently via the switchless call mechanism.

(3) **SGX transitions optimizations.** Previous work [4], [21], [29], [33] circumvents expensive SGX context switches by leveraging threads in and out of the enclave which communicate via shared memory, an approach also implemented by Intel [8].

Tian et al. [28] propose a switchless worker thread scheduling algorithm aimed at maximising worker efficiency so as to maximise performance speedup. `ZC-SWITCHLESS` on the other hand, leverages a scheduling approach aimed at minimising CPU waste while improving application performance relative to a non-switchless system.

VII. CONCLUSION AND FUTURE WORK

In this paper, we highlight the limitations of Intel’s switchless call implementation via experimental analysis. To mitigate the issues raised, we propose `ZC-SWITCHLESS`, an efficient and configless technique to drive the execution of

SGX switchless calls. ZC-SWITCHLESS leverages an in-application scheduler that dynamically configures an optimal number of worker threads which minimises the waste of CPU resources and obviates the performance penalty from misconfigured switchless calls. Our evaluation with a varied set of benchmarks shows that ZC-SWITCHLESS provides good performance while minimising CPU waste.

We will extend ZC-SWITCHLESS by integrating with profiling tools (see §VI), to offer deployers an additional monitoring knob over SGX-enabled systems. Further, while the performance issues with `memcpy` were unexpected, we speculate similar issues might exist in other routines of the `libc`, for which a more in-depth analysis should be dedicated.

ACKNOWLEDGMENTS

This work was supported by the Swiss National Science Foundation under project PersIST (no. 178822) and the VEDLLoT (Very Efficient Deep Learning in IoT) European project (no. 957197).

REFERENCES

- [1] Intel SGX Reference Manual, Section 3.7.6.1. <https://intel.ly/31MrqTQ>.
- [2] Intel SGX SDK, `memcpy`. <https://tinyurl.com/95xcmak>.
- [3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
- [4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’Keeffe, Mark L Stillwell, et al. SCONe: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [5] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference (ATC)*, July 2017.
- [6] Intel Corporation. SDK for Intel software guard extensions. <https://01.org/intel-software-guard-extensions/downloads>.
- [7] Intel Corporation. SDK for Intel software guard extensions ssl. <https://github.com/intel/intel-sgx-ssl>.
- [8] Intel Corporation. Intel software guard extensions developer reference for Linux OS. https://download.01.org/intel-sgx/sgx-linux/2.13/docs/Intel_SGX_Developer_Reference_Linux_2.13_Open_Source.pdf, 2018.
- [9] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(86), 2016.
- [10] Built-in Functions for Memory Model Aware Atomic Operations. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html, 2021.
- [11] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
- [12] Kissdb: simple stupid database. <https://github.com/adamierymenko/kissdb>.
- [13] Dingding Li, Ronghua Lin, Lijie Tang, Hai Liu, and Yong Tang. SGXPool: Improving the performance of enclave creation in the cloud. *Transactions on Emerging Telecommunications Technologies*, 2019.
- [14] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.
- [15] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O’Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzter, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *USENIX Annual Technical Conference (ATC)*, July 2017.
- [16] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: A decade of wasted cores. In *11th European Conference on Computer Systems (EuroSys)*, 2016.
- [17] Mohammad Mahhouk, Nico Weichbrodt, and Rüdiger Kapitza. SGX-oMeter: Open and modular benchmarking for Intel SGX. In *14th European Workshop on Systems Security (EuroSec)*, 2021.
- [18] Giovanni Mazzeo, Sergei Arnautov, Christof Fetzter, and Luigi Romano. SGXTuner: Performance enhancement of Intel SGX applications via stochastic optimization. *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [19] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
- [20] Larry W McVoy, Carl Staelin, et al. Imbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.
- [21] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *11th European Conference on Computer Systems (EuroSys)*, 2016.
- [22] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), jan 2019.
- [23] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.
- [24] OpenSSL Project. Openssl: Cryptography and ssl/tls toolkit. <https://www.openssl.org/>.
- [25] OpenSSL Project. Openssl: Evp-aes-256-cbc. https://www.openssl.org/docs/manmaster/man3/EVP_aes_256_cbc.html.
- [26] The Gramine Project. Graphene performance tuning and analysis. <https://github.com/gramineproject/graphene/blob/master/Documentation/devel/performance.rst#effects-of-system-calls--ocalls>, 2021.
- [27] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [28] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in Intel SGX. In *3rd Workshop on System Software for Trusted Execution (SysTEX)*, 2018.
- [29] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. SGXKernel: A library operating system optimized for Intel SGX. In *Computing Frontiers Conference (CF)*, 2017.
- [30] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522, 2020.
- [31] Intel VTune Profiler: Find and fix performance bottlenecks quickly and realize all the value of your hardware. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>.
- [32] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for Intel SGX enclaves. In *19th International Middleware Conference (Middleware)*, 2018.
- [33] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA ’17*, page 81–93, New York, NY, USA, 2017. Association for Computing Machinery.
- [34] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. Montsalvat: Intel SGX shielding for GraalVM native images. In *22nd International Middleware Conference (Middleware)*, 2021.