# Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves

Ofir Weisse Valeria Bertacco Todd Austin
University of Michigan
oweisse/vale/austin@umich.edu

## ABSTRACT

Intel's SGX secure execution technology allows running computations on secret data using untrusted servers. While recent work showed how to port applications and large-scale computations to run under SGX, the performance implications of using the technology remains an open question. We present the first comprehensive quantitative study to evaluate the performance of SGX. We show that straightforward use of SGX library primitives for calling functions add between 8,200 - 17,000 cycles overhead, compared to 150 cycles of a typical system call. We quantify the performance impact of these library calls and show that in applications with high system calls frequency, such as *memcached*, *openVPN*, and *lighttpd*, which all have high bandwidth network requirements, the performance degradation may be as high as 79%. We investigate the sources of this performance degradation by leveraging a new set of microbenchmarks for SGX-specific operations such as enclave entry-calls and out-calls, and encrypted memory I/O accesses. We leverage the insights we gain from these analyses to design a new SGX interface framework *HotCalls*. *HotCalls* are based on a synchronization spin-lock mechanism and provide a 13-27x speedup over the default interface. It can easily be integrated into existing code, making it a practical solution. Compared to a baseline SGX implementation of *memcached*, *openVPN*, and *lighttpd* - we show that using the new interface boosts the throughput by 2.6-3.7x, and reduces application latency by 62-74%.

## CCS CONCEPTS

• **Security and privacy** → **Security in hardware**; **Systems security**; **Software security engineering**;

## KEYWORDS

SGX, Hardware security, Performance optimization

**ACM Reference format:**
Ofir Weisse Valeria Bertacco Todd Austin University of Michigan . 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080208

## 1 INTRODUCTION

Cloud computing allows lowering the cost of computation and storage, outsourcing the acquisition and maintenance to a third party. Using hardware and software under the control of a third party implies substantial trust: trust that the service provider will not snoop on the data on its servers and will not tamper with the execution flow. Even if the cloud provider can be trusted not to actively snoop or tamper with processed data, users must also trust in the operating system, the virtual machine manager, and the firmware (BIOS & System Management Mode code - SMM). A compromise in the security of any of these layers, by means of remote attack or rogue employee tampering with the hardware, leads to compromising the information and the execution on the cloud.

In 2015, Intel released the Skylake micro-architecture, the first x86 production processor featuring a secure execution technology - Software Guard Extensions (SGX) [5, 23, 35]. This technology allows secure execution in user-space (ring 3) in a container called a secure-enclave, which is shielded from the OS, VMM, and SMM. Ideally, no vulnerability or intentionally malicious code in any of these layers should compromise the confidentiality or the integrity of the secure-enclave. No probing of physical buses outside the processor chip should compromise the security, as the memory is encrypted as well.

Recent work [6, 9, 24, 44, 49] has proposed new frameworks for performing large-scale computations and for porting existing applications to secure-enclaves. Although they provide qualitative discussion about the performance implications of running within an SGX enclave, it remains unclear what specific operations may slow down execution, and by how much. Such quantitative understanding is the corner stone for constructing effective optimization strategies when developing secure-enclaves.

To the best of our knowledge, this work is the first quantitative performance evaluation of SGX, quantifying the overhead of transferring control to and from secure-enclaves and encrypted memory I/O. We give the first taxonomy of the operations involved in using the SGX framework and their costs in machine clock cycles. Based on this analysis, we propose a performance boosting alternative interface to interact with secure-enclaves. We found that the overhead of calling a secure-enclave function is between 8,600 and 17,000 cycles (depending on cache state), compared to 150 cycles for a regular OS syscall [45], and compared to 1,300 cycles for a hypercall in a KVM virtualization solution [15]. We also found that the mechanism allowing secure code to interact with the application or OS outside the enclave incurs between 8,200 and 17,000 overhead cycles (depending on cache state). This is a 54x-113x degradation in performance compared to regular OS calls.

We evaluated several microbenchmarks to estimate the cost of transferring buffers to and from enclaves. While [19] suggests that

*ISCA '17, June 24-28, 2017, Toronto, ON, Canada*

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080208

the Memory Encryption Engine (MEE) adds no more than 12% overhead to the benchmark execution, we found that for our microbenchmarks encryption/decryption may add up to 102% increase in memory access time. On the *mcf* and *libquantum* benchmarks from SPEC 2006 [21], the slowdown was 55% and 420%, respectively.

The overhead of SGX-related calls becomes a significant bottleneck in applications with high system-call frequency. For instance, a database application serving 200,000 requests per second (*e.g., memcached*, as evaluated in Section 6.2) requires at least 200,000 system calls to transfer responses through the network. According to our measurements, each call consumes at least 8,200 cycles, totaling 1,640 million cycles. On a 4 GHz core, this amounts to 41% of the core time spent on merely facilitating the calls, without doing any actual work. Our evaluation of non-trivial applications in Section 6, shows that this is not just a hypothetical problem.

Identifying that context switches used for facilitating system calls are a major bottleneck in SGX applications, we design and implement *HotCalls* - an alternative interface for calling enclave functions and requesting system calls by the enclave. *HotCalls* are based on a spin-lock synchronization mechanism, and provide more than an order of magnitude speedup. Compared to the standard SGX SDK [25] framework, *HotCalls* cost only 620 cycles per system-call in most cases, a 13-27x improvement.

We evaluated the performance of three non-trivial applications within SGX: *openVPN* (encrypted tunnel), *memcached* (memory based database), and *lighttpd* (fast HTTP server), using a straightforward approach to port them into SGX secure-enclaves. We show that, using *HotCalls*, it is possible to improve throughput by 2.6-3.7x and reduce the applications' response latency by 62-74%.

To summarize, we make the following contributions:

- We identify and analyze fundamental operations in SGX technology that have major performance implications. We provide the first comprehensive evaluation of the latency of each such operation, by designing and running a set of microbenchmarks. Based on the microbenchmarks' results, we offer best practices for using SGX when performance is just as important as security.
- Leveraging the insights from the microbenchmarks, we design and implement a new interface to SGX, *HotCalls*, for communication between secure-enclaves and untrusted code. *HotCalls* are 13-27x faster than the existing mechanism provided by the SGX SDK.
- We evaluate the benefit of *HotCalls* on widely used applications: *openVPN*, *memcached*, and *lighttpd*, showing that the throughput of these applications can be improved by a factor of 2.6-3.7x and the response latency can be reduced by 62-74%.

## 2 SGX - BACKGROUND

With the *6th Generation Intel Core* processors, *Skylake*, Intel introduced the Software Guard Extensions (SGX) instruction set that enables the use of a secure execution environment [5, 23, 35]. Similar to ARM TrustZone [3] *secure world*, SGX allows creating a secure execution context, called a *secure-enclave*, protected from the operating system and other user applications. Unlike the *secure world* in ARM TrustZone, the secure context created by SGX has only user-level privileges, and each user application may create several distinct secure-enclaves. A secure-enclave is a reverse sandbox - it protects the user-level software from being compromised by the environment: the operating system, the virtual machine manager, the BIOS (via SMM), and the hardware surrounding the CPU chip. Any of these may be malicious (like adversary OS Iago attacks [10], or hardware cold boot attacks [20]) or compromised (an OS, VMM, or SMM vulnerability [17, 47]). SGX allows clients to securely run software on untrusted servers maintained by a third party such as Amazon cloud computing, Microsoft Azure, or other cloud computing providers.

Guaranteeing confidentiality and integrity of execution on a remote third party server is not trivial. The cloud provider has inherent access to the hardware (memory buses, BIOS image) and the virtual machine management software (VMM), allowing the provider to eavesdrop on the memory contents and the execution flow of all software running on its servers. A security breach may also be introduced by a rogue employee in the cloud company, modifying the BIOS image or patching the VMM software. SGX allows running software operating on secret data in the cloud, without compromising its security. Ideally, it should not be possible for the cloud provider to affect the execution flow of the software, or inspect the secret data being processed, beyond the impact of a potential denial of service attack. SGX secure-enclaves may be used to implement secure databases, software using a secret key to encrypt/decrypt data, or other services processing sensitive data while running on hardware or software under the control of a third party.

The technical details of SGX instructions are detailed in the Software Developer Manual [28]. At boot time, the BIOS defines an area in memory called the Enclave Page Cache (EPC). This is part of the Processor Reserved Memory (PRM) area, which cannot be accessed by any software, regardless of its privileges. The EPC is encrypted by the Memory Encryption Engine (MEE) [19] residing on the processor die. Every processor has two master secrets saved as fused keys, set uniquely at manufacturing time for each individual processor. The first master secret is used to derive memory encryption keys and it is not kept in Intel's records. The second master secret is used to derive a public-private authentication pair, used for attestation, and it is stored in Intel's database. The MEE protects against hardware attacks, trying to snoop on the data when it is in transit to and from memory. The MEE also provides integrity protection, preventing roleback attacks, and protects against malicious modification of the linear-to-physical mapping by a malicious OS or VMM.

The secure context is created by initializing a secure-enclave using the ECREATE [1] instruction. Memory pages containing code and data are copied into the enclave's encrypted memory by invoking the EADD and EEXTEND instructions. This code is referred to as the *trusted code*. The pages added to the enclave are hashed to generate an enclave *measurement*. After all the trusted code and data are transferred into the enclave, the *measurement* is finalized by invoking EINIT. During the attestation process [29], the CPU uses the relevant master secret to sign the measurement and generate a report. The report is passed to a remote client (running on a trusted machine),

[1] SGX supports only two instructions: ENCLU and ENCLS. All operations such as ECREATE, EADD, EINIT, *etc.* are considered leaf functions of ENCLU or ENCLS, but are referred to as simply *instructions* for clarity.

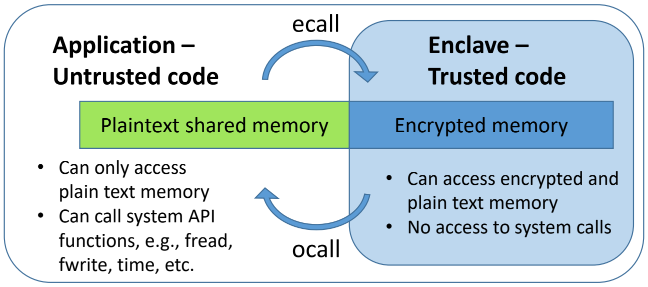
Application –
Untrusted code
ecall
Enclave –
Trusted code
Plaintext shared memory
Encrypted memory
• Can only access plain text memory
• Can call system API functions, e.g., fread, fwrite, time, etc.
ocall
• Can access encrypted and plain text memory
• No access to system calls

**Figure 1: Interaction between the application and the secure-enclave: control is transferred to the enclave via *ecall*; requests for OS API calls are processed via *ocalls*.**

which then contacts Intel's servers to verify that the signature was produced by a genuine Intel processor. The remote client can then provision secret data to be processed by the enclave via a secure channel that is created as part of the attestation process. We refer the interested reader to [13] for additional information.

## 2.1 Application-Enclave Interaction

**Entering the enclave**: Figure 1 illustrates the communication mechanisms between the application and the secure-enclave. After the secure-enclave is initialized, the only way for the *untrusted code* (outside the enclave) to start executing the *trusted code* (inside the enclave) is by invoking the EENTER instruction. EENTER performs the context switch into the enclave, saving the state of the *untrusted code* and restoring the last known state of the *trusted code*. This context switch is conceptually similar to VMENTER and VMEXIT used for virtual machine context switches in Intel's VTX technology [36]. To ease development of secure-enclaves, Intel provides wrapper code, called *ecall* (for entry call), to perform the preparation of the environment and invoke the EENTER instruction [26, 27].

**Accessing external resources**: Because the enclave is *trusted code* running with user-level privileges, *i.e.*, ring 3, it has no access to hardware or other OS resources. In order to gain access to external resources, such as the file system, network, or clock, the enclave must exit to the *untrusted code*. It can do so via the EEXIT instruction. EEXIT performs the reverse context switch and switches back to the *untrusted code*. The wrapper code to do so is called *ocall* (for out call).

**Declaring edge calls**: *ecalls and ocalls* are considered edge functions, as they cause execution to cross security boundaries. The functions' parameters need to be marshalled and copied from encrypted memory to plaintext memory, and vice versa. For the boundary-crossing to be secure, several security checks need to be performed on the call's parameters, particulary in the case that they are pointers. To ease the development of SGX enclaves, Intel provides an edge function creator tool called *edger8r* (pronounced edgerator), to automatically generate secure wrapper code of *ecalls* and *ocalls*. If some of the parameters passed to the edge function are buffers, the specific wrapper code generated depends on whether the buffers are used as input parameters, output, or both.

To automatically generate the ecalls and ocalls code, the programmer must use an SGX-specific syntax to declare the edge functions in an EDL extension file. The declaration includes the parameters each function receives, and their attribute: input, output or both. The *edger8r* then parses the EDL file and generates wrapper glue code for ecalls and ocalls. The glue code consists of two parts: *trusted* and *untrusted*. Our proposed *HotCalls* framework makes use of this generated code to facilitate the calls.

# 3 OVERHEAD OF FUNDAMENTAL OPERATIONS

Inspecting the SGX technical documentation raises several questions with respect to performance:

**What is the overhead of a secure context switch into the enclave, and out of it?** Every access to non-user-space resources, such as files, network, clock, *etc*. requires a context switch to the untrusted code, to perform an OS API call. If the enclave code performs such requests at high frequency, the time spent on the context switches will have a drastic impact on the program's performance. Table 1 describes the microbenchmarks we evaluated. Microbenchmarks 1, 2, 4, and 5 specifically measure context switch latencies.

**What is the cost of passing parameters and data between the application and the secure-enclave?** In order to service these requests, parameters and buffers must be transferred from the secure-enclave to untrusted execution, and vice-versa. Microbenchmarks 3 and 6 in Table 1 measure the cost of transferring data in each direction.

**What is the cost of accessing encrypted memory?** The enclave memory resides in the EPC and it is encrypted. The Memory Encryption Engine (MEE) provides both confidentiality, integrity, and protection from rollback attacks. Providing these security guarantees is expected to come at a performance cost. Microbenchmarks 7 and 8 measure the access time for consecutive memory buffers of various sizes, compared to access times for regular (not encrypted) memory. Microbenchmarks 9 and 10 measure the access time of non-consecutive reads and writes, and estimate the cache miss latency of encrypted memory, compared to that of regular memory.

## 3.1 Experimental Setup

We ran all experiments on a Supermicro server X11SSZ-QF, 64 GB DDR4 RAM @ 2133 MHz, Intel Core I7-6700k 4GHz with 4 hyper-threaded cores (total of 8 logical cores). Dynamic frequency and voltage scaling were disabled; the operating system was the Ubuntu server 14.04 LTS. The SGX SDK version we used is 1.5.80.

**Measuring methodology**: We used the *read time stamp counter* instruction - RDTSCP, to estimate execution time in clock cycles. RDTSCP is a serialized variant of RDTSC, obviating the need to combine the costly CPUID instruction with RDTSC. On production-deployed SGX systems, RDTSC and its variants are not allowed within the enclave, hence all RDTSCP calls must be executed in the untrusted code. We measured RDTSCP to be accurate up to +/- 2 cycles. We ran each microbenchmark for 10 groups of 20,000 runs, totaling 200,000 test executions.

When using RDTSCP to measure cycle count of a user-space operation, it is important to ensure that there are no context switches to the operating system, which would contaminate the measurement. To avoid this contamination, we ran each experiment many times. Since interrupts and context-switches to the OS are infrequent, repeating the experiment multiple times ensure that the majority of the measurements are not interrupted. Moreover, any context-switch to

| # | Micro-benchmark | Description | Median Latency (cycles) |
|---|---|---|---|
| 1 | Ecall (warm cache) | Calling a secure enclave function with no parameters, and immediately returning. See Fig. 2a (solid line). | 8,640 |
| 2 | Ecall (cold cache) | Same as above, the entire cache is flushed between consecutive experiments. See Fig. 2a (dotted line). | 14,170 |
| 3 | Ecall buffer transfer | Calling a secure enclave function, passing 2KB buffer to / from / to&from the enclave. Other buffer sizes are depicted in Fig. 4. | 9,861/11,172/10,827 |
| 4 | Ocall (warm cache) | Exiting the secure enclave to execute an untrusted call. See Fig 2b (solid line) | 8,314 |
| 5 | Ocall (cold cache) | Same as above, the entire cache is flushed between consecutive experiments. See Fig. 2b (dotted line). | 14,160 |
| 6 | Ocall buffer transfer | Calling untrusted code, passing a 2KB buffer to / from / to&from the untrusted code. Other buffer sizes are depicted in Fig. 5. | 9,252 / 11,418 / 9,801 |
| 7 | Reading memory | Consecutively reading from a 2 KB buffer in encrypted/plaintext memory in chunks of 64 bits. Other buffer sizes are depicted in Fig. 6. | 1,124 / 727 |
| 8 | Writing memory | Consecutively writing to a 2 KB buffer in encrypted/plaintext memory, in chunks of 64 bits. Other buffer sizes are depicted in Fig. 7. | 6,875 / 6,458 |
| 9 | Cache load miss | Reading 8 bytes (64 bits) from encrypted/plaintext memory | 400 / 308 |
| 10 | Cache store miss | Writing 8 bytes (64 bits) to encrypted/plaintext | 575 / 481 |

**Table 1: Microbenchmarks targetting fundamental operations using SGX secure enclaves. Every microbenchmark consists of 10 batches of 20,000 experiments, totalling 200,000 measurements. For microbenchmarks involving memory operations, the relevant memory addresses are evicted from the last-level cache prior to every single measurement.**

the OS while the application is executing inside the enclave, causes an *Asynchronous Exit* - AEX, which forces the execution to jump to a known location in the *untrusted code*. We monitored this location in order to count the number of AEX events. Out of 200,000 measurements per micro benchmark, around 200-300 experienced an *Asynchronous Exit*. Hence, we discarded those runs for the sake of performance estimation.

## 3.2 Measuring Ecalls Overhead

When the untrusted code wishes to initiate a trusted function, it does so via an *ecall*. The transition into and out of the trusted code is implemented partially in software, *i.e.,* the SDK, and partially in hardware via the EENTER and EEXIT instructions.

**Software interface:** The ecall is a wrapper to the EENTER instruction. The ecall first locates the enclave with the specified ID, then acquires a read/write lock, finds an available Thread Control Structure (TCS), saves Advanced Vector Extensions (AVX) [18] state, checks for floating point exceptions, and finally calls the EENTER instruction.

**Hardware interface:** The EENTER instruction preforms a secure context-switch as described in Intel's Software Developer Manual (SDM) [28]. Most microcode operations in EENTER involve disabling debugging/tracing mechanisms and defensive checks of enclave management structures: SGX Enclave Control Structure (SECS) and Thread Control Structure (TCS). After validating the SECS and TCS structures, the registers representing the untrusted context (*e.g.,* RAX, RSP *etc.*) are backed up and the enclave context is loaded instead. At the completion of the trusted function execution, EEXIT performs the reverse context switch, and un-suppresses the debugging/tracing mechanisms. As these operations potentially involve sparse encrypted-memory accesses, they may add significant latency (see Section 3.4, microbenchmarks 9,10 in Table 1, and Figure 8).

To measure the latency of performing an *ecall*, we created an empty *ecall*, *i.e.,* a trusted function that receives no parameters and returns no parameters. Since running RDTSCP inside the enclave generates a fault, we can only measure the execution time of entering and exiting the enclave together. The solid line in Figure 2a depicts the cumulative distribution function (CDF) over 200,000 measurements. Over 99.9% of the measurements are between 8,600 and 8,700 cycles. For comparison, [45] estimates a transfer to the OS and back in 150 cycles, and [15] estimates hyper-calls to the hypervisor as taking 1,300 cycles (KVM hypervisor on x86 processor). These measurements reflect performance with a warm cache. Because of the repetitive nature of our tests, the memory structures that are accessed to execute the ecall are in cache for most runs. To eliminate this artifact, we conducted the same experiment, but flushed the entire last level cache (LLC) before each run. The dotted line in Figure 2a depicts the CDF of this experiment: the round trip time of executing an ecall is between 12,500-17,000 cycles, with a median of 14,170 cycles, that is, 83-113x slower than an OS system call.

### 3.2.1 Transferring Memory To/From the Enclave.

When transferring parameters to a secure function, the SDK framework generates code to serialize all the parameters inside a single contiguous data structure. This data structure is in the insecure memory and a pointer to the data structure is transferred to the trusted function. To avoid data leakage from secure memory, the trusted function wrapper verifies that the entire data structure pointed by the pointer is outside the enclave memory.

When dealing with buffers, the programmer can choose among four options: *user_check*, *in*, *out*, and *in&out*. The programmer selects the option when declaring the ecall in the EDL file. The EDL file is written by the programmer with a specific Intel-provided syntax, to declare edge functions (ecalls and ocalls), the parameters they receive, and additional permissions for each edge function. Figure 4 reports the round trip time in cycles of ecall including transferring

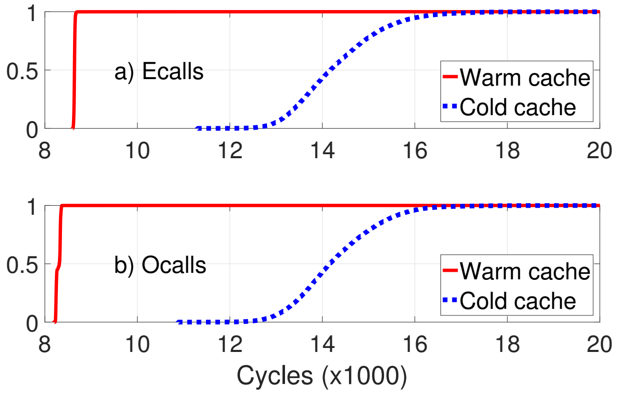

**Figure 2: CDFs of ecalls and ocalls performance. In cold cache experiments the entire 8 MB LLC cache was flushed prior to every experiment, causing relevant data structures and code needed for the ecalls/ocalls to be fetched from memory. a) ecalls: with warm cache, 99.9% of the calls complete between 8,600 and 8,680 cycles. With cold cache 99.9% of the calls take between 12,500 and 17,000 cycles. b) ocalls: without flushing the cache 99.9% of the calls complete in 8,200 - 8,400 cycles. With cold cache 99.9% of the ocalls take between 12,500 and 17,000 cycles.**

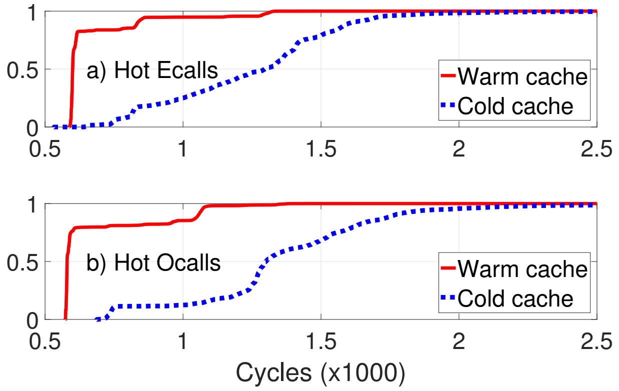

**Figure 3: CDF of HotEcalls and HotOcalls. Over 78% of the calls are executed in less than 620 cycles, and 99.97% are completed within 1,400 cycles. For comparison, the native SGX SDK calling mechanism is 13x-27x times slower. HotCalls' footprint in memory is extremely small, compared to native calls, reducing the chances of cache misses during the HotCall execution.**

buffers to&from the enclave. The cycle count for transferring 2 KB buffers are shown in Table 1, microbenchmark 3.

**Zero copy:** The *user_check* option means that the SGX framework treat the pointer provided as if it were a value parameter. No security checks are carried out to validate if it points to encrypted or regular memory and no copy is done. This is useful if this pointer is pointing to encrypted memory (received earlier from the enclave), or in case of transfer of a pointer, which will be used later by the untrusted code. An example can be a FILE pointer that the enclave will later use in a *fread* call.

**Copying in:** The *in* option tells the *edger8r* tool to generate wrapper code that will allocate memory in the secure memory, according to a size parameter supplied by the untrusted code, and then copy the buffer into the enclave. Memory encryption is transparent to software: memory writes to secure memory are first written unencrypted in cache, and are encrypted by the MEE when the cache line is evicted to RAM. The pointer that will be given to the trusted function implementation will point to a location within the enclave encrypted memory. This is useful especially in cases where a threat of Time-Of-Check-Time-Of-Use attacks (TOCTOU) exists. For example, if the secure-enclave checks a cryptographic signature of a given data, and then uses the (supposedly verified) data for a critical operation, while between the time of check and the time of use the *untrusted code* might have changed the data. In order to measure the accurate latency of transferring new data into the enclave, we removed the buffers inside and out of the enclave from the cache by calling *clflush* on the relevant addresses, before each measurement.

**Copying out:** The *out* option is used when the untrusted code passes a buffer as an output argument, *i.e.*, the trusted code fills the buffer with data. Using the *out* option generates wrapper code that allocates a buffer in secure memory according to a size parameter supplied by the untrusted code, zeroes the entire buffer, and passes it as the pointer for the trusted function. Upon return, this buffer is copied back to the insecure memory. The security reasoning behind zeroing the buffer is to avoid information leakage. Since the buffer is allocated on the secure memory heap, it may initially contain secret data. Should the trusted function fill only part of the out buffer, this secret data will be copied back to the insecure memory, leaking secret data similar to the HeartBleed bug [16]. To prevent data leakage, the SDK zeroes the buffer using a proprietary version of *memset*, which operates byte-wise. This is extremely inefficient on a 64 bit platform, and explains the added latency when using the *out* option.

**Copying in&out:** The *in&out* option is used when the untrusted code passes a buffer as input *and* output argument. In this case the generated wrapper code allocates a buffer in secure memory, copies the data from the insecure memory, and passes the new allocated buffer to the trusted function. Upon return, the buffer is copied back to the insecure buffer, obviating the need to zero the allocated buffer, like in the case of the *out* option. Copying in&out is faster than using just the *out* option, as the *memcpy* used by the SDK is more efficient than the byte-wise *memset* used in the *out* option.

## 3.3 Measuring Ocalls Overhead

When the secure code requires access to external resources, such as network, files, clock *etc.,* it needs to invoke out-calls.

**Software interface:** Similar to ecalls, ocalls are declared in the EDL file. The *edger8r* tool provided in the SGX SDK generates trusted and untrusted glue code. The trusted code marshals data structures, performs security checks on pointers values, and then executes the EEXIT instruction. The untrusted code organizes the input arguments, calls the requested OS API call (*fopen, send etc.*), marshals data to be returned to the enclave (the ocall output arguments) and executes the ERESUME instruction to resume execution of the trusted code.

**Hardware interface:** The operations performed by EEXIT are as described in Section 3.2. The operations performed by ERESUME

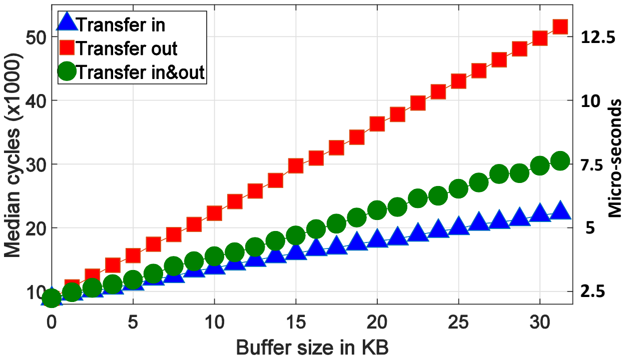

**Figure 4: Latency of ecall + transferring a buffer in/out/in&out. Transferring a buffer out is extremely taxing due to the inefficient *memset* implementation in SGX SDK.**

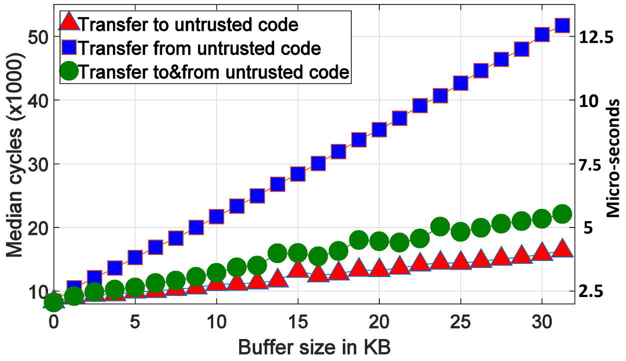

**Figure 5: Latency of ocall + transferring a buffer to/from/to&from untrusted memory. Transferring from untrusted code has high latency, due to (redundant) zeroing of the buffer in the untrusted memory with the inefficient *memset*, provided by the SDK.**

instruction are described in Intel's SDM [28]. ERESUME performs similar operations as EENTER, but resumes execution of the trusted code from the instruction after EEXIT.

As in ecalls, pointer parameters can be marked in the EDL file as *user_check, in, out, in&out* to instruct the *edger8r* how to generate the code. Figure 2b shows the round trip latency for performing an ocall. Figure 5 shows the performance implication of using the in/out/in&out options when transferring buffers to/from ocalls.

**Zero-copy**: As before, *user_check* entails zero-copy and no checks. This is useful when passing pointers that are provided by the OS, such as a file pointer.

**Copying out**: contrary to ecalls, in the case of ocalls, the *in* option means "into the ocall", *i.e.,* from the secure memory *out* to the insecure memory. The wrapper code verifies that the pointer points to a location within the enclave. It then allocates memory on the insecure stack according to a size parameter supplied by the enclave (no use of *malloc* here). Finally, it copies the buffer to the insecure memory. Upon re-entry to the enclave the allocated memory is freed by unwinding the insecure stack.

**Copying in**: The *out* option stands for "out of the ocall, into the enclave". The wrapper code allocates memory on the insecure stack, according to a size parameter supplied by the enclave. The newly allocated buffer in the insecure buffer is then zeroed. After the ocall itself returns, the buffer is copied back into secure memory. In our opinion, zeroing the buffer in the insecure memory has no security benefit. The untrusted code can access this memory anyway, prior to invoking the latest ecall. As mentioned before, zeroing the buffer is carried out via *memset*, which the SDK implements as byte-wise zeroing. This is extremely inefficient, and explains why the *out* option is much slower than the *in&out* option. We observe that zeroing a buffer in the plaintext memory does not add a security benefit, and thus this operation can be removed. In Section 6, we evaluate the impact of our *No-Redundant-Zeroing* approach on common applications.

## 3.4 Measuring Memory Access Overhead

The enclave code can access both regular memory, and enclave memory. The enclave memory resides within the Enclave Page Cache (EPC) and is encrypted by the Memory Encryption Engine (MEE), which resides on the processor's die and is described in [19]. The MEE provides confidentiality, integrity, and anti-roll-back protections for the entire EPC. These guarantees are provided by maintaining an *integrity tree*, with its root stored on the processor's die. A full walk of the tree involves several memory accesses. Therefore, a "MEE cache" is used to prevent significant performance costs, when accessing nearby memory addresses.

The work presented in [19] also analyzes the potential performance degradation of using encrypted memory, and evaluates the worst case benchmark to exhibit a 12% overhead when using encrypted memory. However, our measurements shows substantially higher overhead, as will be detailed shortly.

**Consecutive buffer access**: Figures 6, 7 plots the memory read/write times when accessing encrypted and plaintext memory, for different buffer sizes. Entries 1 and 8 in Table 1 list the results of reading and writing 2 KB buffers. The memory read & write microbenchmark consists of accessing 8-bytes (64 bit) aligned words of consecutive addresses, for different buffer sizes. The buffers were flushed out of the last level cache (LLC) before each experiment, and *mfence* was called before the final call to RDTSCP, to ensure the operations had completed. When measuring write latency, the experiment was completed by flushing the buffer from cache via *clflush* followed by *mfence* prior to calling the final RDTSCP.

**Cache misses**: To estimate cache-load-miss and cache-store-miss latency, we performed the same read and write experiments, accessing only the first 8 bytes (64 bit) of an address which is aligned to the cache line size (64 bytes on the tested machine). The cache line was evicted from the LLC before each measurement. The first four bars of Figure 8 shows the results on our micro benchmarks for reading and writing in encrypted memory, and for cache miss penalties. The cycle counts are detailed in lines 9,10 of Table 1. Cache-load misses and cache-store-misses are 30% and 19.5% slower when accessing encrypted memory vs. plaintext memory.

**SPEC 2006 memory intensive benchmarks**: To test the effect of more complicated memory access patterns, we selected three highly memory-intensive benchmarks from SPEC 2006 [21]: *mcf, libquantum,* and *astar*. Figure 8 compares the latency of the memory

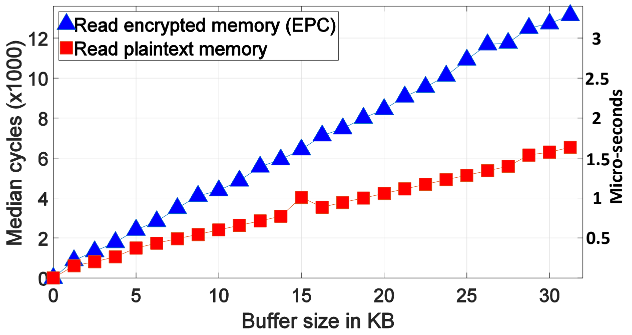

**Figure 6: Latency of consecutive memory reads, for encrypted and plaintext memory. All buffers were evicted from the cache prior to every experiment. The overhead of reading encrypted memory of sizes 2,4,8,16,32 KB is 54.5%, 68%, 71%,94%,102%, respectively.**

microbenchmarks mentioned and the selected benchmarks from SPEC 2006. Previous work published by Intel [19] measures several benchmarks from SPEC 2006, showing that, in the worse case, the encryption overhead is roughly 12%. In our measurements, *mcf* runs 55% slower in the SGX enclave, and *libquantum* runs 5.2x slower. A likely explanation for the extreme slowdown in the case of *libquantum*, is that it required 96 MB of memory, while the entire Enclave Page Cache (EPC) is 93 MB. This forces paging out encrypted memory pages, which requires further SGX operations.

## 3.5 Lessons Learned

We now discuss insights from the measurements, which can be used by developers designing secure-enclaves to devise optimization strategies. These insights were also instrumental in our design of *HotCalls*, as we discuss in Section 4.

**Cost of ecalls & ocalls**: Compared to regular OS syscalls, an ecall is 54x more cycles at best (8,200 vs 150) when the cache is warm, and 83-113x at worst when cold (12,500-17,000 vs 150). If an application has high call rate, for example, 100,000 calls per second, on a 4 GHz core the ocalls will consume 20-40% of the execution time. The applications evaluated in Section 6 exhibited more than 200,000 calls per second, as detailed in Table 2. Our solution, *HotCalls*, proposes an alternative calling mechanism that reduces this latency to as low as 620 cycles per call, which is 13-27x faster than the default ecalls and ocalls mechanism.

**Ocalls vs. Ecalls**: Ocalls may execute slightly faster than ecalls. Transferring buffers from the enclave to the untrusted application is faster using ocalls: 9,252 cycles for ocalls *in* vs. 11,712 cycles for ecalls *out* (2 KB buffers). This insight may lead to an optimization strategy of using ocalls to receive data from the enclave, rather then delivering it via an output parameter with ecalls.

**Cost of memory access**: Write accesses of encrypted memory incur 6.5-19.5% overhead, and read accesses incur 30-102% overhead, depending on buffer size. For software that is memory read intensive, we can estimate the impact on throughput in the following way: without encryption, N memory reads are carried out in time $T$, and with encryption in $1.5T$ (assuming 2 KB buffers). Thus, we can expect that encryption slow down throughput to $\frac{N}{1.5T}/\frac{N}{1T} = 66\%$ throughput, not accounting for any other SGX performance impact.

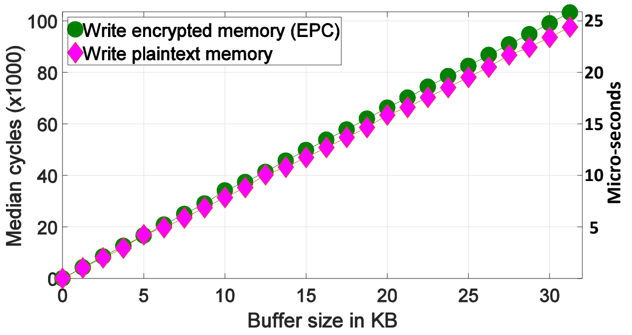

**Figure 7: Latency of consecutive memory writes, for encrypted and plaintext memory. All buffers were evicted from the cache prior to every experiment. The overhead of writing encrypted memory is roughly around 6% for all buffer sizes above 1 KB.**

**Selecting the right transfer method**: For both ecalls and ocalls, the *out* option is extremely inefficient. This is due to the inefficient implementation of *memset* in the SDK, used for zeroing buffers. Assuming that the untrusted code has nothing to hide from the enclave, it is more efficient to use the *in&out* option instead. Despite the fact that the buffer will be redundantly copied to the secure memory, this will save 885 / 1617 cycles for ecalls / ocalls in the case of a 2 KB buffer.

**Opting for user_check**: If the enclave just dumps data to the output buffer, then there is no threat of the untrusted code modifying the data while the enclave code is processing it. In that case it is preferable to use the *user_check/zero-copy* option, and have the enclave directly write data to un-encrypted memory. This will save about 3,000 cycles on a 2 KB buffer (11,712 vs 8,640 cycles). Developers should be careful when using this option. If the secure-enclave performs encryption or decryption in place, having access to the partially-processed buffer during the encryption/decryption process may lead to exposing the secret key.

**Further optimizations**: The latency incurred in buffer transferring is incurred by *memcpy* and *memset*. The SGX stdlib implementation of *memset* is operating on memory byte-wise, which is extremely inefficient on a 64 bit processor. Using a more optimized version of *memset* may significantly improve performance. Additionally, when large buffers need to be transferred, it may by beneficial to use optimized versions of *memcpy*, utilizing Advanced Vector Extensions (AVX) [18] instructions which are able to copy words larger than 64-bits efficiently. Intel may wish to include this optimization in future versions of the SGX SDK.

# 4 HOTCALLS: AN OPTIMIZED SGX INTERFACE

Motivated by the fact that using the default SDK calling mechanism may lead to a 113x slowdown, we now present *HotCalls*, an alternative mechanism to perform ecalls and ocalls, leading to an order of magnitude performance improvement. Compared to 8,200-17,000 cycles required for SDK ecalls/ocalls, *HotCalls* can be as fast as 620 cycles. While SGX calls rely on expensive secure context switches, *HotCalls* operations are based on using shared un-encrypted memory.

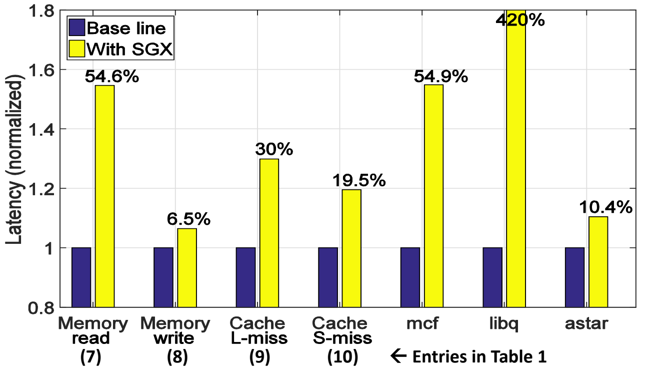

**Figure 8: The overhead of memory encryption on memory-access speed: L and S stand for Load and Store. Memory reads and writes are of consecutive 2 KB buffers. *mcf*, *libquantum* (libq), and *astar* are memory intensive benchmarks from SPEC 2006.**

## 4.1 HotCalls Architecture

Edge calls in SGX are context-switch operations, similar to VMENTER and VMEXIT used for virtual machine context switches in Intel's VTX technology [36]. Previous work exists on optimizing communication mechanisms between hardware and software (interrupt handlers), and virtual machine manager and guest operating systems (hyper-calls or VM-exits). An approach that has shown to be effective is avoiding the software context switch by using shared memory as a communication channel and dedicating a thread to poll for new messages. This has been tried in Linux NAPI to optimize access to hardware [43] and in virtualization scenarios to eliminate the need for an expensive context switch [31, 34]. We take a similar approach and propose an architecture that consists of a *requester* and a *responder*, communicating via un-encrypted shared memory.

Figure 9 illustrates this architecture, where the enclave code is the requester, and the untrusted code is the responder. The requester is the party requesting a call, while the responder is an *On Call* thread, standing by, waiting for a call. It does so by constantly polling a shared memory location. Synchronization of the shared memory is provided using a spin-lock. When the requester makes a call, it acquires the spin-lock and checks a shared Boolean variable to verify that the responder is not currently busy. If the responder is available, the requester copies relevant data to unencrypted shared memory, and points to that data via the **data* pointer. The code to encapsulate parameters within the *data* structure is the same code used by the SDK ecalls/ocalls mechanism, that is automatically generated by the *edger8r* tool. To support more than one specific call, *e.g.,* read, write *etc.,* the requester specifies the ID of the call it is requesting. This is an entry ID to a function call table, known to the responder. The call table approach is similar to the SDK implementation of ecalls and ocalls. Once the data pointer and the requested call ID are in place, the requester signals "go" to the responder, by marking the responder as busy, and releases the lock. The responder is constantly monitoring the same shared memory and executes the relevant call when requested.

## 4.2 Practical Considerations

**Spin-lock**: Use of standard POSIX MUTEX is not possible, as it requires calling upon the operating system services, defeating the entire purpose of HotCalls. Synchronization techniques using MONITOR/MWAIT instruction entails several thousands of cycles, similar to regular SGX calls [4]. The SGX SDK provides a spin-lock implementation as *sgx_spin_lock*. This is a straightforward busy-wait implementation and does not relate to SGX, so it can be used by both the enclave and the untrusted code.

**Minimizing self-contention**: To ensure that both the requester and responder get a chance to acquire the spin-lock, *PAUSE* instructions are added after releasing the lock. This gives a chance to other threads to try and acquire the spin-lock. The *PAUSE* instruction was designed by Intel specifically to improve performance in spin-lock busy wait loops, by minimizing memory order violations of speculative loads, and also to help reduce power consumption.

**Maximizing utilization**: As the responder is constantly monitoring the shared memory, it is effectively using 100% of the logical core. The utilization can be considered as the amount of time the responder is spending on *ExecuteCall* vs. the time spent on acquiring the lock and checking the Boolean flag value. This utilization can potentially be improved by sharing the responder thread with several requesters.

**Preventing starvation**: Contention of several requesters on the responder can cause the requester to loop many times before it acquires the lock and the responder is available. The maximum worst-case wait time is therefore potentially unbounded. As a mitigation for this potential starvation, the requester can set a timeout - a maximum number of times to check if the responder is available. If the timeout expires, the requester can fall back to using regular SDK calls. In our experiments and evaluation of applications, we set this timeout to 10, and it never expired. Nevertheless, we find this mechanism vital for producing reliable code.

**Conserving resources at idle times**: When there are not many calls, the responder wastes CPU resources, constantly polling shared memory. To conserve resources during idle times, a timeout counter can be set. The counter is decremented when there is no request waiting (step 2 in Fig. 9, right side), and reset when a request arrives (in step 3). When the timeout counter expires, the responder set a *sleep* flag, and goes to wait on a conditional variable (POSIX pthread_cond, or *sgx_thread_cond*). The requester notices the *sleep* flag is set and signals the condition variable before issuing the request.

**Marshalling parameters to be transferred to the function**: The SDK generates code for packing ecall/ocall parameters. The framework we built for our evaluation automatically uses this code to pack/unpack the parameters and copy buffers if needed. This solution allows us to also avoid redundant buffer zeroing when transferring data from the untrusted code, without compromising security. The performance implications of removing the redundant *memset* will be detailed in the next section.

## 4.3 Empirical Evaluation of HotCalls

Our implementation of HotCalls consists of 115 lines of code. Similar to the microbenchmarks described in Section 3, we performed

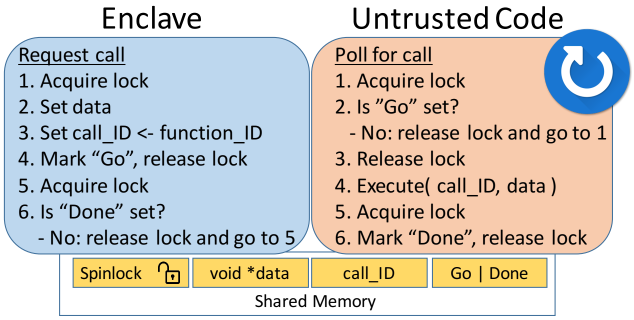

**Figure 9: HotCall architecture. The secure-enclave requests a call by signaling a request via a shared variable in un-encrypted memory, together with the ID of the requested function. The responder thread in the untrusted side continuously polls the shared memory to check if a call request has been made.**

10 batches of 20,000 measurements each, totalling 200,000 measurements. Figure 3 shows the CDF of the latency of HotEcalls and HotOcalls. In both cases, more than 78% of the calls took less than 620 cycles (warm cache). Over 99.97% of the calls took fewer than 1,400 cycles. For comparison, the ecalls/ocalls mechanism provided by the SGX SDK requires 8,200-17,000 cycles.

## 4.4 Implications of Using an Additional Core

The benefit of using an additional logical thread to utilize HotCalls can be analyzed from two perspectives: application's throughput and overall power consumption. Analysis of both perspectives depends on whether HotCalls increases the application's throughput by more than a factor of 2, as explained below.

**Throughput:** When considering optimizing an application with HotCalls, the overall throughput increase by using HotCalls should be compared to the potential benefit of simply adding an additional worker thread. This is not always possible, as some applications are developed in a single thread. The additional extra thread cannot increase the overall throughput by more than a factor of 2. Hence, HotCalls are preferred over adding an additional worker thread, when it more than double the throughput.

**Power consumption:** When the responder is idle, or underutilized, it issues the *PAUSE* instruction in a loop, therefore it is not expected to consume much power. If the responder is idle for relatively long periods of time it can conserve power by waiting on a conditional variable, releasing the core resources, as suggested in Section 4.2. When the responder is busy, and the overall throughput increases by more than a factor of 2 (as for all the applications evaluated in Section 6) even if the power consumption doubles, the power per given throughput unit is still more efficient when using HotCalls.

# 5 SECURITY ANALYSIS

In order to assess to security implications of using HotCalls we examine the modifications which may affect the trusted code running inside the enclave.

**Using shared plaintext memory for communication:** Our HotCalls technique for passing data structures between the enclave and the untrusted code is no less secure than the SGX SDK's mechanism. HotCalls source code for marshalling data structures between the enclave and the untrusted code is *the same code* used by the SDK's ecalls and ocalls implementation, generated by the *edger8r* tool. Any manipulation possible on data, which was marshalled by HotCalls, is also possible when using the SDK's ecalls and ocalls.

**Attacks on the *data* pointer:** Any security breach possible via manipulating the *data* pointer used by HotCalls is also possible on the pointer passed by the SDK's implementation of ecalls and ocalls. For ecalls, the responder (inside the enclave) identifies the request for a call, and passes the *data* pointer to the original function created by the *edger8r* tool. From then on the source code is identical to the default SDK implementation, including all security checks on the pointer in untrusted memory, and the copying of relevant buffers by using *in* or *out* modifiers (see Section 3.2.1).

In the case of ocalls, the HotOcall wrapper in the trusted code is almost identical to the original ocall code generated by the *edger8r* tool. The only difference is replacing the call to the SDK function *sgx_ocall*, responsible for invoking the EEXIT instruction, with code requesting a *HotCall*, similar to the "Request call" function illustration in Figure 9.

**Requesting a function via call_ID:** The technique of setting a function number in shared memory is also utilized by the SDK. The "call_ID" in HotCalls is comparable to the "ocall_index" variable used by the SDK. Any manipulation on "call_ID" is also possible on the "ocall_index" passed by the SDK to the untrusted ocall. Such manipulation to the "call_ID" or "ocall_index" will cause the untrusted code to execute a wrong function, hence no new vulnerability is introduced.

**Using the spin-lock located in shared memory:** Tampering with the synchronization provided by the spin-lock will either cause a denial of service (DoS) due to a deadlock, which is out of the SGX threat model or will cause multiple threads to access the same data in plaintext memory at the same time. Similar to HotCalls, the SDK's ecalls and ocalls involve pointers in untrusted memory, which are accessible to the adversary in the SGX threat model. Therefore, the adversary can manipulate these pointers to cause multiple threads to access the same memory simultaneously, whether the threads are executing trusted or untrusted code.

# 6 EVALUATING HOTCALLS ON APPLICATIONS

To evaluate the performance impact of SGX on complex software we chose three applications: memcached, openVPN, and lighttpd. Figures 10 and 11 show the throughput and latency of these applications in normal execution and when running inside an SGX enclave. Each application, at its peak utilization, is performing hundreds of thousands of Linux API calls, each second. Table 2 lists a breakdown of the most frequent API calls in each application, and the execution time spent on facilitating the calls. Using HotCalls and the *No-Redundant-Zeroing* approach we were able to reach a 2.6-3.7x throughput boost, compared to the unoptimized implementation, and reduce the average response latency by 62-74%.

## 6.1 Efficient Application Porting

SGX Enclaves shield the code running within them from the rest of the system. Communication to and from the enclave is carried out via defining edge functions (ecalls and ocalls). Previous work

| Application | Frequent Calls<br>(Calls x1000 / second) | Total Calls | Core Time |
|---|---|---|---|
| Memcached | read(66.5), sendmsg(66.5)<br>RunEnclaveFucntion(66.5) | 200K | 42% |
| OpenVPN | poll(87), time(87), getpid(13.6),<br>write(30), recvfrom(30),<br>read(13.6) sendto(13.6) | 275K | 57% |
| Lighttpd | read(49),fcntl(25),<br>epoll_ctl(25), close(25),<br>setsockopt(25), __fxstat64(25)<br>inet_ntop(12),accept(12),<br>inet_addr(12),ioctl(12),<br>__open64_2(12), sendfile64(12)<br>shutdown(12),writev(12) | 270K | 56% |

**Table 2: Number of API calls (in thousands per second) in non-optimized memcached, openVPN and lighttptd, running inside a SGX secure-enclave. Each ocall, including both software and hardware interfaces (see Section 3.3), takes roughly 8,300 cycles (assuming a warm cache). On a 4 GHz core, the execution time is thus $N_{calls} \cdot 8,300$ / $(4 \cdot 10^9)$, which is listed in *Core Time* column.**

[6, 9, 40] argued that it is desirable to port applications into an SGX enclave as a whole, minimizing the number of code modifications. Any change to production-grade software may introduce new bugs and potentially new security vulnerabilities. We determined to take a similar approach as the baseline SGX implementation of the applications under test. The main ecall was defined as simply calling the application's original *main* function. We used the makefile provided by the SGX SDK [25] as a guideline for building the enclave shared object. Any OS API call used by the application will result in an ocall.

**Identifying API calls**: Any call to a function outside the codebase results in an *undefined reference* error at link time. Examples of such functions are *fopen, fread, time, socket* and so forth. Each application under test had between 93-144 such *undefined references*. For each function, it is required to generate a wrapper function that will be executed inside the enclave, and an EDL declaration of an ocall, describing the nature of the arguments (input, output, size of buffers), and finally a landing function in the untrusted code which will call the relevant OS API. We developed a framework to identify the undefined references and generate the needed trusted/untrusted wrapper code. As it is sometimes hard to infer programmatically the input/output nature of an argument, and its size in memory, our framework allows adding exceptions by hand. The wrapper code allows adding counters that estimate the number of calls per second of each function. A breakdown of the most frequent calls per application is presented in Table 2.

**Marshalling data structures**: In HotCalls, only a void pointer is transferred between the enclave and the application (and vice versa). The *edger8r* tool provided by the SDK generates marshalling code to pack/unpack parameters from a structure, perform the necessary security checks, and copy buffers. Our framework automatically extracts the marshalling code generated by the *edger8r* tool, to generate wrapper code to be used with HotCalls.

**Corner case API calls**: Some API calls require special attention. A new thread creation with *pthread_create* eventually calls a function inside the enclave, using a pointer provided to the call in *start_routine*. If *start_routine* points to code residing inside the enclave, this call will fail. Therefore, we created a new *ecall* edge function RunEnclaveFunction, which receives a pointer to code inside the enclave and jumps to it, in the same way *pthread_create* would.

The structures *pthred_mutex_t, pthread_cond_t* are used as synchronization mechanisms. The SDK provides alternatives to these structures: *sgx_thred_mutex, sgx_thread_cond*, that should be used instead. The SDK also provides matching locking/unlocking and waiting/signaling operations. Whenever these synchronization mechanisms were used, we replaced them with the SGX SDK alternatives.

## 6.2 Memcached

Memcached is a key-value RAM database. It is typically used as a caching layer between web-servers and the back end database to boost performance. Memcached is widely deployed: *e.g.*, Facebook, Zynga, Twitter, Youtube [33, 42]. As suggested by *CryptDB* [39], it is desirable to protect the confidentiality and integrity of a database by encrypting it. Completely porting the database application to run within an SGX enclave offers similar security guarantees as *CryptDB*, with minimal development effort.

A thorough workload analysis of deployed memcached instances is provided in [8]. The majority of the requests are under 2KB for the size of the requested value. As a caching layer, memcached performance is measured by the number of requests handled per second and the latency for replying to each request, as observed by the requesting client.

We evaluated memcached version 1.4.31, on the same platform described in Section 3. We tested the performance with *memtier benchmark* [41], a tool developed by Redis Labs to evaluate memcached performance. The benchmark used the binary protocol, the ratio of SET:GET was set to 1:1, and the data size of the payload was set to 2KB. A total of 4 million requests were issued, from 4 concurrent threads. We ran memcached with a single thread. To avoid hindering the performance due to link-capacity we ran both memcached and memtier-benchmark on the same machine - using the loopback as the network interface. The original memcached source was able to service, on average, 316,500 requests per second, with average response latency of 0.63 milliseconds. Unoptimized porting of memcached into the SGX enclave, as described in Section 6.1, reduced the throughput to 66,500 requests per second, and the average response latency to 2.97 milliseconds, a 79% reduction in serviced requests and a 4.7x increase in latency.

Porting memcached to run inside an enclave exposed 93 external API references. Table 2 provides a breakdown of the most frequent API calls: *read* and *sendmsg*. On average *read* and *sendmsg* are called from within the enclave 66,500 times per second. Memcached utilizes *libevent* to wait on a socket, and receives a callback when new data is available. Since the callback function is inside the enclave, it requires invoking an ecall, RunEnclaveFunction. RunEnclaveFunction is called on average also 66,500 times per second. To ensure that the counters do not interfere with the performance measurement, we

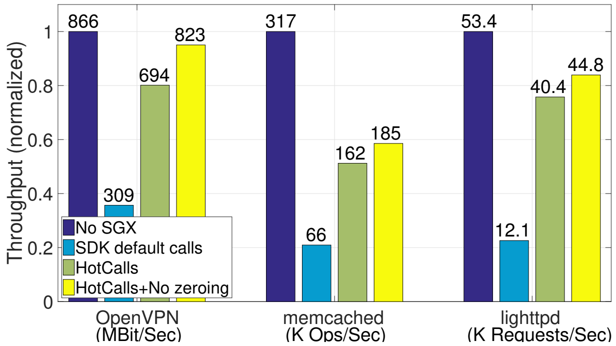

**Figure 10: Optimizing throughput with *HotCalls* and No-Redundant-Zeroing. The measurements are normalized to running without SGX. Memcached, by nature, is a memory-intensive application, and therefore optimization is limited by the performance of the memory encryption engine.**

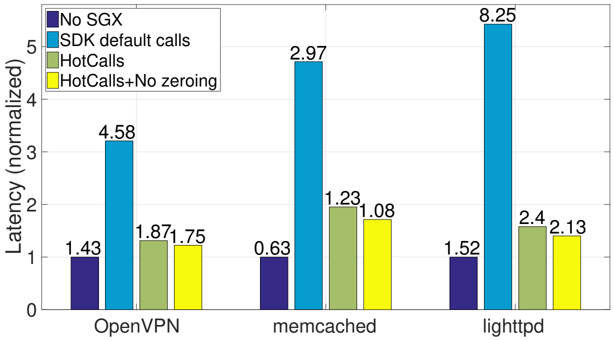

**Figure 11: Optimizing latency with *HotCalls* and No-Redundant-Zeroing. The values above the bars are in milliseconds: For openVPN the values are the average ping round-trip time. For memcached and lighttp the values are the server's average response latency.**

repeated the benchmark with the counting code removed. Measured results were similar.

There are total of 199,500 edge calls per second. As ecall requires at least 8,600 cycles, and ocall requires at least 8,200 cycles, this sums up to over 1.7 billion cycles per second spent merely on transferring control between SGX and untrusted code, not accounting for memory transfers. On a 4GHz core, this is 42% of the core time spent on SGX context switching.

**Results**: Utilizing HotCalls for *read*, *sendmsg*, and RunEnclaveFunction increases the throughput to 162,000 requests serviced per second, and reduces the response latency to 1.23 milliseconds, a 2.4X increase in throughput and 58% decrease in latency. Figures 10 and 11 depicts HotCalls impact on throughput and latency. The *read* API call is receiving a buffer from the network, therefore using the *out* category of ocalls. Removing the redundant zeroing performed by the SGX generated code increased throughput to 185,000 requests per second, and reduced response latency to 1.08 milliseconds. This is 2.8x increase compared to SGX SDK calls, and a 64% reduction in latency.

**Fundamental limitation**: Even with HotCalls, the throughput is still only around 60% of the non-SGX baseline. Memcached is by nature memory intensive. The memory accesses are uniform across the memory-stored database, leading to poor spatial locality, and therefore suffer from many cache misses. As suggested by the memory access microbenchmark and *mcf* benchmark in Section 3, memory access latency of 2KB buffers may be slowed down up to 55%, potentially causing throughput degradation up to 35%, just accounting for memory access overhead. This is inherent to using encrypted memory. Recent work has suggested a speculative loading mechanism to improve the performance of encrypted memory [22], and may improve the performance of memory intensive applications such as memcached.

## 6.3 OpenVPN

openVPN [38] is a well known widely used open-source Virtual Private Network (VPN) solution that provides secure encrypted tunnels between two endpoints connected to the Internet. openVPN utilizes OpenSSL library [37] as the cryptographic implementation. Compromising the secret keys used by openVPN compromises the security of the tunnel, potentially allowing an outsider to inspect tunnel communication or inject traffic. Therefore, it may be desirable to port openVPN into an SGX enclave to protect encryption and authentication keys.

We evaluated openVPN version 2.3.12, using OpenSSL version 1.0.2. We created a secure tunnel between the SGX machine specified in Section 3, and a desktop machine Intel NUC 5i7RYH with 16 GB of DDR3, running Ubuntu Desktop 16.04 LTS, connected in a 1 Gbit/sec link. To evaluate the throughput, we used *iperf3* [30]. We ran *iperf3* for 60 seconds to estimate the actual maximum TCP bandwidth between the desktop and the SGX machine and found it to be 935 Mbit /sec. We then ran *iperf3* over the openVPN tunnel (without modification) and measured a TCP bandwidth of 866 Mbit /sec, showing that the Ethernet link is not fully saturated. Unoptimized porting of openVPN into the SGX enclave, as described in Section 6.1, reduced the TCP bandwidth to 309 Mbit/sec, a 64% decrease. We estimated the round trip latency using a flood ping, sending 1 million requests, with preload of 100 requests before waiting for the response. For native openVPN, the average round trip latency was 1.427 milliseconds. The SGX implementation increased the average latency to 4.579 milliseconds, a 3.2x increase.

Porting openVPN to run inside an enclave exposed 131 external API references. Table 2 provides a breakdown of the most frequent API calls, totaling roughly 275,000 ocalls per second, wasting 57% of the core time merely on SGX context switches. A surprisingly frequent API call is *getpid*. This call is invoked by OpenSSL whenever a cryptographic context is called. Other frequent API calls are *inet_ntop, inet_addr*. These are utility functions, which could be implemented inside the secure-enclave to reduce the number of ocalls and improve performance.

**Results**: Utilizing HotCalls for all seven frequent API calls, increases the bandwidth to 694 Mbit/sec, and reduces the round trip latency to 1.873 milliseconds, a 2.25x increase in bandwidth and 60% decrease in latency. Results are depicted in Figures 10 and 11. The calls *recvfrom and read* receive a buffer from the untrusted code, therefore using the SDK *out* category of ocalls. Removing the redundant zeroing performed by the SGX generated code increases bandwidth to 823 Mbit/sec and reduces the round trip latency to 1.747 milliseconds: a 2.66x bandwidth increase and 62% decrease in latency.

## 6.4 lighttpd

lighttpd [32] is an open source, light-weight web server optimized for speed and serving many concurrent requests. It runs single-threaded in a single process. We evaluated lighttpd version 1.4.41. We evaluated its performance using *http_load* [1]. The measurement consisted of 100 concurrent clients connections, fetching a total of 1 million 20 KB pages. The connections were over the local loopback to maximize available link bandwidth. Unmodified *lighttpd* was able to serve an average of 53,400 pages per second, with average response latency of 1.52 milliseconds.

Unoptimized porting of *lighttpd* into SGX enclave, as described in Section 6.1, reduces the number of requests per second to 12,100, and increases the response latency to 8.25 milliseconds, a 77% decrease in throughput and 5.4x increase in latency. Porting *lighttpd* to run within an enclave exposed 144 external API references. Table 2 gives a breakdown of the most frequent API calls, totaling in roughly 270,000 ocalls per second, costing 56% of the core time. The API calls *inet_ntop*, and *inet_addr* don't require OS involvement and can be implemented inside the enclave, reducing by 9% the number of ocalls. The rest of the calls, however, do require access to external OS resources, and can not be optimized into the enclave.

**Results**: Utilizing HotCalls for all 14 frequent API calls, increases the throughput to 40,400 requests per second and reduces the response latency to 2.40 milliseconds, a 3.3x increase in throughput and 70% decrease in latency. Results are depicted in Figures 10 and 11. The calls *read* and *inet_ntop* receive a buffer from the untrusted code, therefore using the SDK *out* category of ocalls. Removing the redundant zeroing performed by the SGX generated-code increases throughput to 44,800 requests per second, and reduced the response latency to 2.13 milliseconds, a 3.7x increase in throughput and 74% decrease in response latency compared to an unoptimized SGX implementation.

# 7 RELATED WORK

**SGX related sources**: The SGX official documentation is in Intel's Software Developer Manual [28]. Explanation of the technology can be found in "Intel SGX explained" [13], and in a presentation given at Black Hat [2]. None of these documents provide specific measurements for the various operations supported by SGX, nor suggest optimization strategies.

**Similar technologies**: *Sanctum* [14] is an alternative secure execution technology for the RISC-V [7, 46] open processor. *ARM Trustzone* is ARM's secure execution solution [3], allowing co-existence of a *secure world* OS, in parallel to the *normal world* OS to manage sensitive data and operations. SecureME [12] introduced a potential architectural support for protecting user space applications' memory from other applications and privileged code such as the OS, VMM or SMM code. Overshadow [11] proposed a VMM enforced mechanism to offer similar protection, but only against other applications and the OS.

**Previous work on SGX**: Drawbridge [40] is a solution for moving most of the OS operations into user space to improve security. Haven [9] makes use of Drawbridge in order to port entire applications with most of their OS-support needs inside an SGX enclave. VC3 [44] is an implementation of map-reduce across multiple servers, running on SGX secure-enclaves. SCONE [6] shows how to run docker containers inside a secure-enclave, and suggests techniques to improve the container performance. These solutions are very effective but are mainly applicable to applications running inside docker. Ryoan [24] offers a framework that allows one enclave to trust another enclave from a different provider, to receive secret data, guaranteeing that it would not be leaked. A possible side-channel attack against SGX is described in [48], exploiting the leakage of memory access patterns in page granularity, *i.e.,* 4 KB. Although all the above mention performance implications of using SGX secure-enclaves, no specific measurement of operations is provided. Therefore, it was not previously clear what optimization strategies should be made to improve performance when developing enclaves.

**Related optimization work**: The approach of polling for events, instead of being called, was suggested for use with hardware in Linux NAPI [43], to minimize the need for a context switch to interrupt handlers. In the virtualization context, this approach has been investigated in [31, 43] as a way to accelerate accessing hardware or virtualized hardware, saving the costly context switch between guest OS and hypervisor. The implementation of *HotCalls*, suggested in this paper, is inspired by these approaches.

# 8 CONCLUSION

Identifying the bottlenecks of Intel's SGX technology is an important step to make secure computation practical. Leveraging the insights from the microbenchmarks allows developers to focus the optimization effort effectively. *HotCalls* is an alternative calling interface for SGX secure-enclaves, which optimizes the calling latency by a factor of 13-27x. Using *HotCalls*, the throughput of common applications can be boosted by up to 3.7x, and the response latency can be reduced by up to 74%. In order for security-enhancing technologies to be prevalent, they need to be practical. The best practices discussed in this work and *HotCalls* are an important step in making SGX both a secure and a practical solution.

# ACKNOWLEDGEMENTS

This work was supported in part by C-FAR, one of the six STARnet Centers, sponsored by MARCO and DARPA. The authors would also like to acknowledge Matthew Hicks for his suggestions while shaping the ideas introduced in this work, and Jeremy Erickson for his assistance and feedback.

# REFERENCES

[1] *http_load - multiprocessing http test client*. http://acme.com/software/http_load/.
[2] *SGX Secure Enclaves in Practice: Security and Crypto Review. Black Hat*. https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-\Crypto-Review.pdf.
[3] Tiago Alves and Don Felton. 2004. TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems.
[4] Nikos Anastopoulos and Nectarios Koziris. 2008. Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors. In *Proc. of IEEE IPDPS*.

[5] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology For CPU Based Attestation and sealing. In *Proc. of HASP*.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, and others. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proc. of OSDI*.

[7] Krste Asanović and David A Patterson. 2014. *Instruction Sets Should be Free: The Case for RISC-V*. Technical Report. University of California at Berkeley, http://www. eecs. berkeley. edu/Pubs/TechRpts/2014/EECS-2014-146. pdf.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS Performance Evaluation Review*.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. of ACM TOCS*.

[10] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proc. of ACM SIGARCH Computer Architecture News*.

[11] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan Ports. 2008. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. of ACM SIGARCH Computer Architecture News*.

[12] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. 2011. SecureME: a Hardware-Software Approach to Full System Security. In *Proc. of ACM ICS*.

[13] Victor Costan and Srinivas Devadas. *Intel SGX explained*. Technical Report. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[14] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. of USENIX Security*.

[15] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. 2016. ARM Virtualization: Performance and Architectural Implications. In *Proc. of ISCA*.

[16] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and others. The Matter of HeartBleed. In *Proc. of ACM IMC*.

[17] Shawn Embleton, Sherri Sparks, and Cliff C Zou. 2013. SMM Rootkits: A New Breed of OS Independent Malware. In *Security and Communication Networks*. Wiley Online Library.

[18] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. 2008. Intel AVX: New frontiers in performance improvements and energy efficiency.

[19] Shay Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. Intel Corporation.

[20] J Alex Halderman, Seth Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel Feldman, Jacob Appelbaum, and Edward Felten. 2009. Lest We Remember: Cold-Boot Attacks on Encryption Keys. In *Communications of the ACM*.

[21] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. In *Proc. of ACM SIGARCH Computer Architecture News*.

[22] Andrew Douglas Hilton, BC Lee, and TS Lehman. 2016. PoisonIvy: Safe Speculation for Secure Memory. In *Proc. of ACM MICRO*.

[23] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proc. of HASP*.

[24] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. of OSDI*.

[25] Intel. *Intel SGX Software Development Kit (SDK)*. Intel. https://software.intel.com/en-us/sgx-sdk.

[26] Intel. *Intel Software Guard Extensions SDK for Linux OS*. Intel. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[27] Intel. *Intel Software Guard Extensions SDK for Windows OS*. Intel. https://software.intel.com/sites/default/files/managed/b4/cf/Intel-SGX-SDK-Developer-Reference-for-Windows-OS.pdf.

[28] Intel. *Software Developer Manual, chapters 37-43*. Intel. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[29] Intel. *Software Guard Extensions: EPID Provisioning and Attestation Services*. Intel. https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf.

[30] Iperf. *A tool for active measurements of the maximum achievable bandwidth on IP networks*. Iperf. https://iperf.fr/.

[31] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. 2007. Rearchitecting VMMs for Multicore Systems: The Sidecore Approach. In *Proc. of WIOSCA*. Citeseer.

[32] lighttpd. *An open-source web server optimized for speed-critical environments*. lighttpd. https://www.lighttpd.net/.

[33] Kevin Lim, David Meisner, Ali Saidi, Parthasarathy Ranganathan, and Thomas Wenisch. 2013. Thin Servers With Smart Pipes: Designing SoC Accelerators for Memcached. In *Proc. of ACM SIGARCH Computer Architecture News*.

[34] Jiuxing Liu and Bulent Abali. 2009. Virtualization polling engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization. In *Proc. of ACM ICS*.

[35] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution.. In *Proc. of HASP*.

[36] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. 2006. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization.. In *Intel Technology Journal*.

[37] OpenSSL. *Cryptography and SSL/TLS Toolkit*. OpenSSL. https://www.openssl.org/.

[38] OpenVPN. *An open source SSL VPN solution*. OpenVPN. https://openvpn.net/.

[39] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. of SOSP*.

[40] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proc. of ACM SIGPLAN*.

[41] Redis Labs. *memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached*. Redis Labs. https://https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached#.WBz0PNzHXeA.

[42] Paul Saab. 2008. Scaling Memcached at Facebook.

[43] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. 2001. Beyond Softnet. In *Proc. of USENIX ALSC*.

[44] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proc. of IEEE S&P*.

[45] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. of OSDI*.

[46] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The RISC-V Instruction Set Manual, Volume I: Base user-level ISA. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* (2011).

[47] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking SMM Memory via Intel CPU Cache Poisoning. In *Invisible Things Lab*.

[48] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of IEEE S&P*.

[49] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proc. of NSDI*.