Performance Principles for Trusted Computing with Intel SGX

Anders T. Gjerdrum[1], Robert Pettersen[1], Håvard D. Johansen[1], and Dag Johansen[1]

[1]UIT The Arctic University of Norway,
Dept. Computer Science, Tromsø, Norway
anders.t.gjerdrum@uit.no

Abstract. Cloud providers offering Software-as-a-Service (SaaS) are increasingly being trusted by customers to store sensitive data. Companies often monetize such personal data through curation and analysis, providing customers with personalized application experiences and targeted advertisements. Personal data is often accompanied by strict privacy and security policies, requiring data processing to be governed by non-trivial enforcement mechanisms. Moreover, to offset the cost of hosting the potentially large amounts of data privately, SaaS companies even employ Infrastructure-as-a-Service (IaaS) cloud providers not under the direct supervision of the administrative entity responsible for the data. Intel Software Guard Extensions (SGX) is a recent trusted computing technology that can mitigate some of these privacy and security concerns through the remote attestation of computations, establishing trust on hardware residing outside the administrative domain. This paper investigates and demonstrates the added cost of using SGX, and further argues that great care must be taken when designing system software in order to avoid the performance penalty incurred by trusted computing. We describe these costs and present eight specific principles that application authors should follow to increase the performance of their trusted computing systems.

Keywords: Privacy, Security, Cloud Computing, Trusted Computing, Performance

1 Introduction

Pervasive computing and the ongoing Internet of Things (IoT) revolution have led to many new mobile recording and sensory devices that record ever more facets of our daily lives. Captured data is often analyzed and stored by complex ecosystems of cloud hosted services. Storing and analyzing large amounts of data are non-trivial problems. Handling personal data such as smart home monitoring systems and health tracking, only adds the to this complexity as data processing might be governed by strict privacy requirements [7].

The curation and analysis of privacy sensitive personal data on third-party cloud providers necessitate the design of a new Software-as-a-Service (SaaS) architecture that is able to enforce rigid privacy and security policies [10] throughout the entire software stack, including the underlying cloud provided Infrastructure-as-a-Service (IaaS). Commodity hardware components for trusted computing have been available for some time [13, 16], but the functionality of existing solutions has been limited to establishing trust and guarantees on the integrity of running software, and rudimentary support for secure code execution (e.g., Intel Trusted Execution Technology).

In 2015, Intel introduced the Software Guard Extensions [1] as part of their sixth generation Intel Core processor micro architecture (codenamed Skylake). Together with complementary efforts by ARM and AMD, SGX is making general trusted computing a commodity, providing confidentiality, integrity and attestation of code and data running on untrusted third-party platforms. SGX is able to deter multiple different software and physical attacks by establishing secure execution environments, or enclaves, of trusted code and data segments inside individual CPUs. While SGX is an iterative technology building upon previous efforts, it is more general in functionality allowing code execution inside enclaves at native processor speeds, a significant performance improvement over previous efforts. SGX is designed with backwards compatibility in mind, allowing developers to port sensitive logic from existing legacy applications into secure enclaves. These properties make SGX a compelling technology for cloud based SaaS hosting privacy sensitive data on untrusted third-party cloud providers. SGX is a proprietary technology and prior knowledge of its characteristics is mostly based on limited documentation by Intel. In particular, little is known about the performance of the computing primitives comprising SGX and how developers should best utilize these to maximize application performance.

This paper provide an in-depth investigation into key performance traits of the Intel SGX platform. We provide a performance analysis of its low-level mechanisms and primitives, and describe several non-obvious idiosyncrasies related to threading, context switching, and memory footprint. From our observations, we derive 1 principles for developing more efficient software on this platform.

The remainder of this paper is structured as follows: Section 2 outlines the relevant parts of the SGX micro architecture while Section 3 outlines the details of our micro benchmarks. Section 4 provides an informed discussion of our findings and a set of derived principles intended for developers of trusted computing systems. Section 5 details relevant work before concluding remarks.

2 Intel Software Guard Extensions (SGX)

Intel's new general trusted computing platform enables the execution of code on untrusted third-parties at native processor speed. Moreover, the platform preserves the confidentiality and integrity of code and data segments running inside what is referred to as *enclaves*. This section details the core mechanisms

comprising SGX, building a foundation for the performance analysis detailed in Section 3.

2.1 Enclave Creation

Enclave code and data are distributed to runtime systems in form of a shared library which is bundled together with what the developer reference refers to as the SIGSTRUCT data structure. During the compilation of an enclave, a hash, or measurement, of each code and data segment executable within the shared library is computed and stored together with a signature generated by the developers private key. This bundle is then distributed to the target third-party platform together with the corresponding public key. During initialization, the signature is verified against the public key and the measurement is recalculated and compared with the corresponding value inside the SIGSTRUCT. If the signature matches that of the public key and the integrity of the code and data segments are preserved, the enclave is allowed to execute. This establishes a guarantee that only the expected enclave code and data from the expected enclave author are successfully able to run on the third-party.

2.2 Entry and Exit

Regular application threads are able to enter secure enclaves by invoking the EENTER instruction on a particular logical core. The thread then performs a controlled jump into the enclave code, similar in operation to a call-gate. Threads can only enter enclaves from privilege level 3 (user level).

Software interrupts are prohibited when running in *encalve mode*. As a consequence, no system calls are allowed within enclaves. Applications requesting access to common Operating System (OS) resources such as IO, must therefore explicitly exit the enclave prior to invocation. The application developer explicitly defines these transitions and, in the presence of a potentially malicious OS, all such transitions, parameters to these and responses must be carefully validated.

Although threads cannot be instantiated in enclave mode, SGX allows multiple threads to enter the same enclave and execute concurrently. For each logical core executing inside a particular enclave, a Thread Control Structure (TCS) is required to keep track of thread specific context. Before instantiation, these data structures must be provisioned and stored in the Enclave Page Cache (EPC), comprising pages explicitly set aside for enclaves. The TCS contains an OENTRY field specifying the entry point for the thread, loaded into the instruction pointer upon entry. Stack regions are not explicitly handled by the SGX microcode, however, as Costan and Devadas [5] state, the stack pointer is expected to be set to a region of memory fully contained within the enclave during entry transition. Parameters input to the developer-specified entry points are marshaled and, once the transition is done, copied into enclave memory from untrusted memory. Although not handled by SGX directly, parameter marshaling and stack pointer

manipulation are managed under the hood by the SDK implementation which most application authors will use for enclave developement.

Threads may transition out of enclaves by means of two different mechanisms, either synchronously trough the explicit EEXIT instruction, or asynchronously by service of a hardware interrupt. Synchronous exists will cause the thread to leave enclave mode, restoring the execution context to its content prior to enclave entry. Asynchronous Enclave Exit (AEX) is caused by a hardware interrupt such as a page fault event. In this case all threads executing on the logical core affected by the interrupt must exit the enclave and trap down to the kernel in order to service the fault. Before exit, the execution context for all logical cores executing within the enclave is saved and subsequently cleared to avoid leaking information to the untrusted OS. When the page fault has been serviced, the ERESUME instruction restores the context and the enclave resumes execution.

2.3 Enclave Memory

During boot-up of the CPU, a contiguous region of memory called Processor Reserved Memory (PRM) is set aside from regular DRAM. Divided into 4 kB pages, only accessible inside the enclave or directly by the SGX instructions, this region of memory is collectively referred to as the EPC. Any attempts to either read or write EPC memory from both privileged level system software or regular user level applications are ignored. Moreover, any Direct Memory Access (DMA) request to this region is explicitly prohibited, deterring physical attacks on the system bus by potentially malicious peripheral devices. Confidentiality is achieved through Intels Memory Encryption Engine (MEE), further preventing physical memory inspection attacks as enclave data is encrypted at the CPU package boundary on the system bus right after the L3 cache.

Much the same as regular virtual memory, EPC pages are also managed by the OS. However, these are handled indirectly through SGX instructions as EPC memory is not directly accessible. The OS is responsible for assigning pages to enclaves and evict unused pages to regular DRAM. Through memory management, the physical limit of 128 MB is evaded by swapping EPC pages and as such there is no practical limit to the size of enclaves. The integrity and liveness of pages being evicted are guarded by an axillary data structure also contained within the PRM, called the Enclave Page Cache Map (EPCM). The EPCM maintains the mappings between virtual and physical addresses of PRM memory. Moreover, it maintains for each page an integrity check and a liveness challenge vector. These precautions guard against a malicious OS trying to subvert an enclave by either manipulating the address translation, explicitly manipulating pages, or serving old pages back to the enclave (replay attacks). In this memory model, only one enclave can claim ownership of a particular page at one given moment, and as a consequence shared memory between enclaves is prohibited. Enclaves are however allowed to read and write directly to untrusted DRAM inside the host process' address space, and therefore two enclaves residing within the same host process are able to share untrusted memory.

Because stale address translations may be exploited to subvert enclave integrity, the processor performs a coarse-grained Translation Lookaside Buffer (TLB) shootdown for each page subject to eviction. Given a page fault event on a particular thread executing inside an enclave, all threads executing on that same logical core must perform an AEX, as described in Section 2.2. In order to avoid information leakage stemming from memory access patterns inside enclaves, the lowermost 12 bits of the faulting address, stored in the CR2 registry are cleared. SGX instructions explicitly support batching up to 16 page evictions together at a time, thus curtailing the cost of AEX for each page fault inside an enclave.

2.4 Enclave Initialization

SGX allows the creation of multiple, mutually distrusting enclaves, on the same hardware instance. These can reside in either a single process' address space or multiple. To instantiate enclave system software the OS, on behalf of the application, invokes the ECREATE instruction. This causes the underlying microcode implementation to allocate a new EPC page for the SGX Enclave Control Structure (SECS), identifying each enclave and storing per-enclave operational metadata. Moreover, physical pages are mapped to enclave SECS through the EPCM structure. Before initialization is complete, each separate code and data segment must be added to enclave memory explicitly through the EADD instructions. Similarly, each TCS is added for each logical core expected to execute within the enclave. Once this process is complete the OS issues the EINIT instruction which finalizes initialization and compares the enclave measurement observed to the contents of the SIGSTRUCT. Upon completion, a launch token is generated by a special pre-provisioned enclave trusted by Intel, at which point the enclave is considered fully initialized. Once this process is completed, no further memory page allocations may happen. Intels revised specifications for SGX version 2 includes the possibility for dynamic paging support by means of the EEXTEND command. However, we refrain from further comment, as hardware supporting these features have not yet been released at the time of writing.

Inversly, during teardown of an enclave, the opposite operation is performed. The OS tags each page as invalid, by issuing the EREMOVE instruction. Prior to this, SGX verifies for each page that no threads attributed to that page are executing inside the enclave. Lastly, the SECS is destroyed once all pages referring to it through the EPCM are themselves deallocated.

2.5 Enclave Attestation

In order for applications to securely host privacy-sensitive software components on platforms outside of their administrative domain, we need to establish trust. This can be achieved through remote attestation, a process in which the remote party proves its correctness to the initiator. Assuming an enclave has been created and initialized as outlined above on an untrusted platform, the entity wishing to establish trust with this enclave issues a request for proof. The code

inside this enclave then requests a *Quote* from the hardware, which consists of the enclave measurement, in addition to a signature from the hardware platform key. This quote is then sent to the requesting party which can themselves validate the measurement compared to the expected provisioned enclave. Lastly, the quote is sent to Intel for verification through their *Intel Attestation Service*, which validates the signature against their own private key. These two in combination prove to the requesting party that the expected code and data segments are running on a valid SGX-enabled platform.

3 Experiments

The next generation of SaaS systems should be designed from the ground up to utilize trusted computing features in a performance optimal way. Therefore, we conduct a series of micro benchmark experiments on a SGX-enabled CPU to fully understand the micro architectural cost of trusted computing on commodity hardware. Our experimental setup consists of a Dell Optiplex workstation with an Intel Core i5-6500 CPU @ 3.20 GHz with four logical cores and 2×8 GB of DDR3 DIMM DRAM. Dynamic frequency scaling, Intel Speedstep and CStates are disabled throughout our experiments to avoid inaccuracies. We set the PRM size to its maximum allowed 128 MB to measure the peak theoretical performance of the platform. Our experiments ran on Ubuntu 14.04 using the open source kernel module by Intel implementing OS support for SGX[1]. Furthermore, this module has been modified with instrumentation in order to also capture the operational cost from the system perspective. Based on our knowledge regarding SGX, we have derived a set of benchmarks conjectured to capture core aspects of the trusted computing platform. It is worth noting that for all our experiments, more iterations did not yield a lower deviation. We attribute this to noise generated by the rest of the system that while subtle, becomes significant at fine-grained time intervals.

The current generation of SGX does not support the use of the RDTSC instruction or any other native timing facilities inside enclaves. Intel has later released a microcode update to counter this problem, allowing for the RTDSC instruction to execute inside enclaves. We are however unsuccessful, at the time of writing, in obtaining a firmware update specific to our SKU through the correct OEM. Measurements performed throughout the experiments must therefore exit the enclave for each point in time. Consequently, all measurements therefore include the time taken to enter and exit the enclave, described as the sequence of events detailed in Figure 1.

3.1 Entry and Exit Costs

With SGX, SaaS applications are able to influence the size of their Trusted Computing Base (TCB) by partitioning application logic between trusted and

[1] https://github.com/01org/linux-sgx-driver

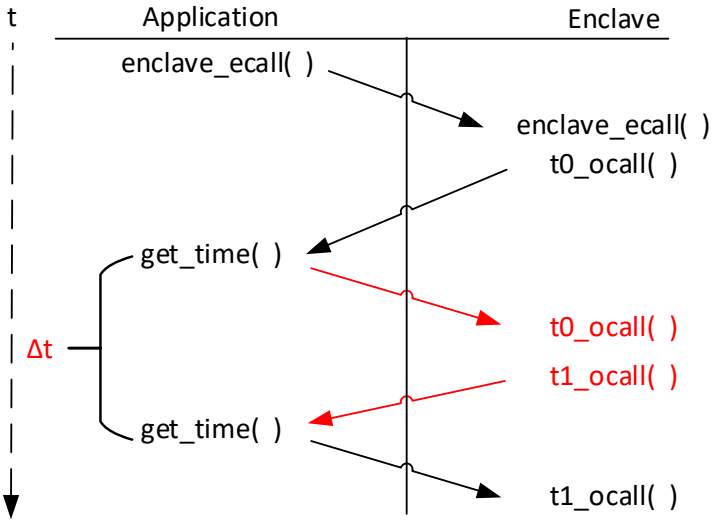

Fig. 1. Sequence of events involved in measuring time spent inside enclaves [8].

untrusted execution domains. In order to quantify any potential performance trade-off, we examine the associated cost of enclave transitions. An optimal application arrangement should conciser the following trade-off depending on the transition cost: A high cost of transition would necessitate a reduction in the overall amount of transitions and mediating this cost will increase the amount of logic residing within the enclave, thus expanding the TCB. A prominent example at one end of the spectrum is Heaven [3], in which an entire library OS is placed within a secure enclave. Furthermore, details in the Intel Software Developer Manual[2] suggest that that the cost of entering an enclave should also factor in the cost of argument data copied as part of the transition into the enclave. Therefore, if the cost of data input to an enclave is high, only data requiring explicit confidentiality and trust should be placed within the enclave.

Figure 2 depicts the measured cost in millisecond latency, as a function of increasing buffer sizes. The cost of entering an enclave is observed to increase linearly with the size of the buffer input as the argument. It is worth mentioning that only buffer input to the enclave is considered. The experiment does not include output buffers or return values from enclaves.

Hosting a buffer inside enclave memory requires that the enclave heap is sufficiently large. Since enclave sizes are final after initialization, we set the heap size to be equally large for all iterations of the experiment. From the graph, we observe that the baseline cost of entering an enclave quickly becomes insignificant

[2] https://software.intel.com/en-us/articles/intel-sdm

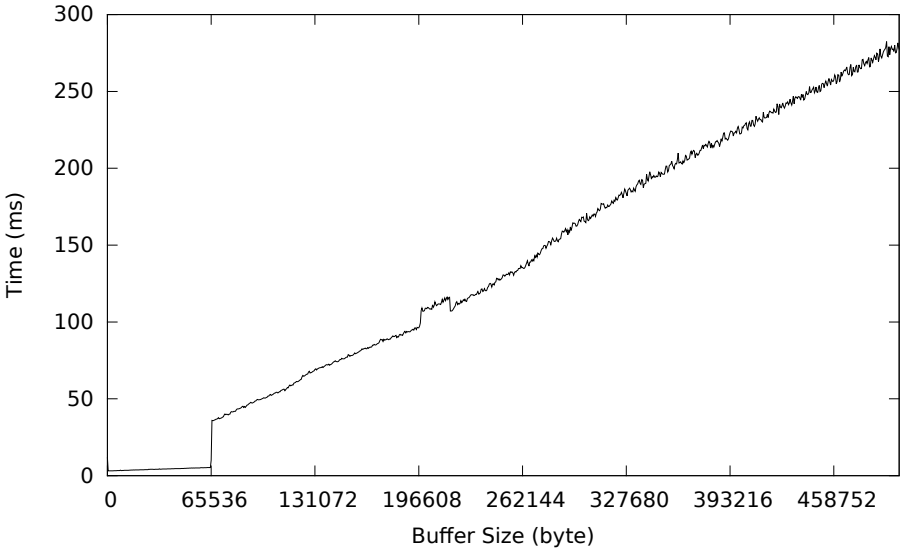

Fig. 2. Enclave transition cost as a function of buffer size [8].

as the buffer size increases. This behavior is not surprising, as the overall cost includes the cost of copying memory into the enclaves which invokes the MEE for each page written to the enclave. A curious observation, however, is the fact that the baseline cost only increases linearly for buffers larger than 64 kB. This could be explained by enclaves less than 64 kB being fully provisioned into EPC memory at startup. Whereas for large buffers the cost may be attributed to lazily loaded enclave memory, triggering page faults during the buffer copy operation. This aspect is explored in detail in the following experiment.

3.2 Paging

Another aspect to consider in the application trade off between TCB and enclave transition cost, is the fact that an increase in TCB would cause an increase in PRM consumption. Moreover, as stated in Section 2.3, PRM is a a fairly limited resource compared to regular memory and the depletion of this resource will cause system software to evict EPC pages more aggressively. As such, any application utilizing SGX should consider carefully the cost of enclave memory management, more specifically the cost of page swapping between EPC and regular DRAM. Figure 3 illustrates this overhead as observed by both the OS kernel and inside the enclave.

The y-axis is the discrete cost in nano seconds, while the x-axis is time elapsed into the experiment. The SGX kernel module has been instrumented to measure the latency of page eviction denoted by the red dots, and the total time spent in the page fault handler, represented by the black solid line.

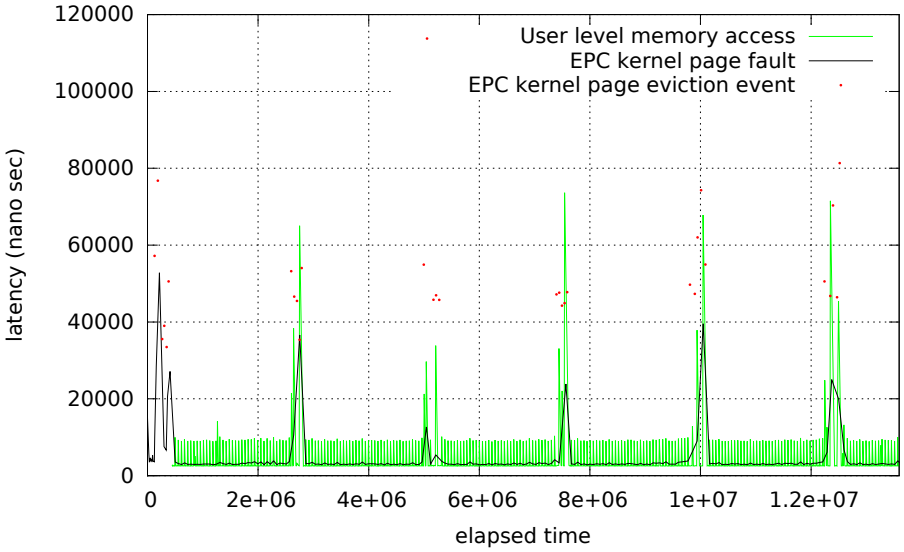

Fig. 3. Paging overhead in nano seconds as a function of time elapsed while writing sequentially to enclave memory [8].

From the enclave perspective, the green line denotes the user level instrumentation and represents time spent writing to a particular address in enclave memory. As mentioned in the experimental introduction, measurement primitives are unavailable inside enclaves, and all user level measurements therefore include the cost of entry and exit, including a 4 byte word as parameter input each way.

To induce page faults, the experimental enclave heap size is set to 256 MB, double that of the of the physical PRM size made available by hardware. Moreover, we invoke write operations on addresses located within each 4 kB page sequentially along the allocated memory address space inside the enclave.

Recall from Section 2 that all memory for a particular enclave must be allocated prior to initialization. We observe from Figure 3 that prior to enclave startup, a cluster of page fault events occur at the beginning of the experiment, corresponding with our prior observations. The system is attempting to allocate memory for an enclave of 256 MB while only being physically backed by 128 MB of EPC memory.

The events occurring at user level can easily be correlated with the observations made in the page fault handler. For each increase in latency observed from inside the enclave, a corresponding cluster of evictions occur in the page fault handler. Moreover, the total time spent in the page fault handler coincides with the write overhead observed at user level. Parts of the overhead can be attributed to the fact that page faults cause AEX events to occur for each logical core executing within the enclave, as detailed in Section 2.

Moreover, we observe that the SGX kernel module is behaving conservatively in terms of page evictions, and is not exhausting EPC memory resources. As detailed in Section 2, the 12 lower bits of the virtual page fault address are cleared by SGX before exiting the enclave and trapping down to the page fault handler. Hence, system software is not able to make any algorithmic assumptions about memory access patterns to optimize page assignment. Furthermore, liveness challenge vector data might also be evicted out of EPC memory, causing a cascade of page loads to occur from DRAM. As a side note, this experiment only uses a single thread, and all page evictions only interrupt this single thread.

In light of the prior discovery, high performance applications should consider tuning the SGX page fault handler to their particular use case, given that the application is able to predict a specific access pattern. Moreover, regardless of access pattern the SGX page fault handler should be optimized to allow exhaustive use of EPC, such that applications running inside enclaves may be less affected by page faults in high memory footprint scenarios.

The initial setup of enclaves will retain large amounts of the pages in EPC memory, alleviating the overhead of paging in certain situations. Moreover, this reduces the execution overhead caused by threads performing AEX. Given that the cost of enclave setup is still a large factor, by the prior statements, it might be advantageous for application developers to pre-provision enclaves.

3.3 Enclave provisioning

Modular programming and componentized system organization are paradigms commonly used in modern distributed systems. Applications consisting of possibly multiple trust domains and third-party open source components should separate the unit of failure and trust to reduce the overall system impact.

By enabling the creation of mutually distrusting enclaves, SGX is able to support a modular application architecture. Section 2 explains how enclaves might communicate with the untrusted application through well defined interfaces, lending itself to compartmentalization of software into separate enclaves. To capture the cost of using SGX through the scope of a modular software architecture, Figure 4 illustrates the cost in terms of provisioning latency as a function of enclaves created simultaneously for differently sized enclaves. We observe that the added cost of enclave creation increases linearly for all sized enclaves, becoming significant for enclaves larger than 256 kB. As detailed in Section 2, enclaves are created by allocating each page of code and data to the enclave prior to initialization. During this experiment we observed a significant amount of page faults further attributing to the creation cost. This is expected as the size of enclaves combined with number of instances increases above that of the physically available PRM. Our observations about buffers less than 64 kB from Section 3.1 still stands, as we observe that the provisioning cost for enclaves less than 64 kB is nearly identical.

To offset the latency of creation for enclave instances, real-time applications should consider pre-provisioning them. However, as prior experiments show co-

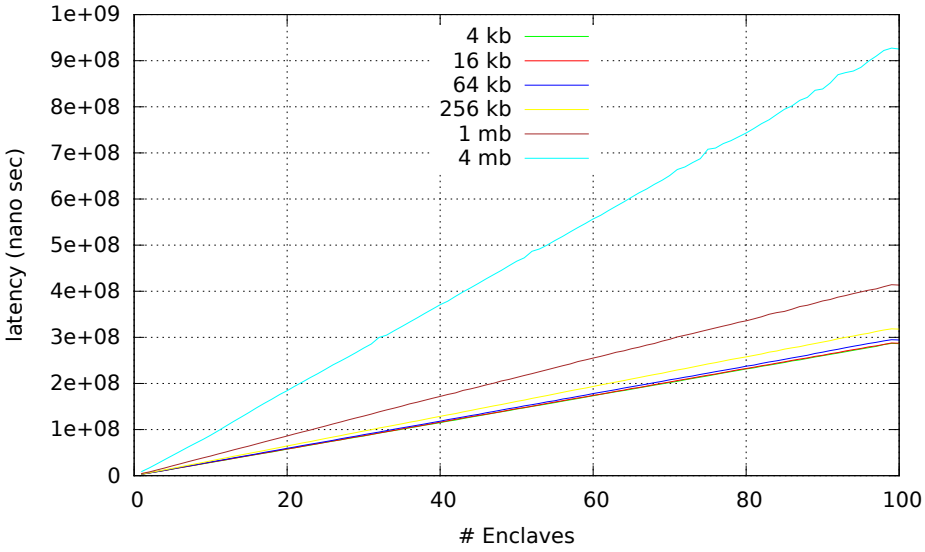

Fig. 4. Latency as a function of number of enclaves created simultaneously, for differing sizes of enclaves [8].

locating multiple enclaves in EPC memory might result in additional cost if the memory footprint is large enough.

3.4 Multithreading

The curation and analysis of large amounts of data use concurrency as a measure to speed up processing of data elements. This is especially true for *embarrassingly parallel* computations. One example is the *distinct count* aggregate operation, where a large corpus of data is sectioned into buckets and where each can be counted in parallel. Such computations require parallelism built into the runtime. Fortunately, SGX provides the ability to run multithreaded operations inside the same enclave. However, implementation details reveal that applications with high memory footprint might suffer from extensive page faults, which can act as a barrier and in the worst cases degrade performance significantly. Furthermore, as we argued earlier, applications with multiple tenants might want to isolate analytics execution into separate enclaves, and it is therefore important to consider how threads are delegated inside of enclaves.

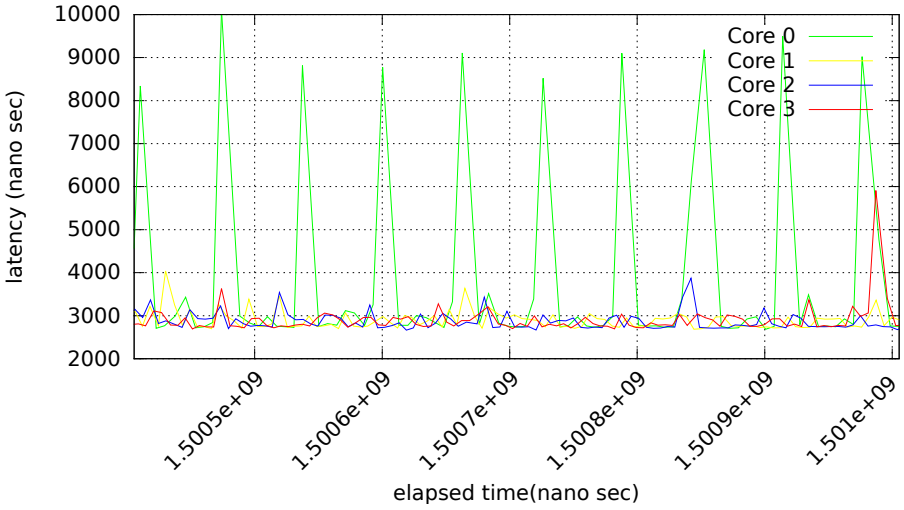

Fig. 5. Execution overhead for multiple threads running on separate logical cores, with page fault events occurring.

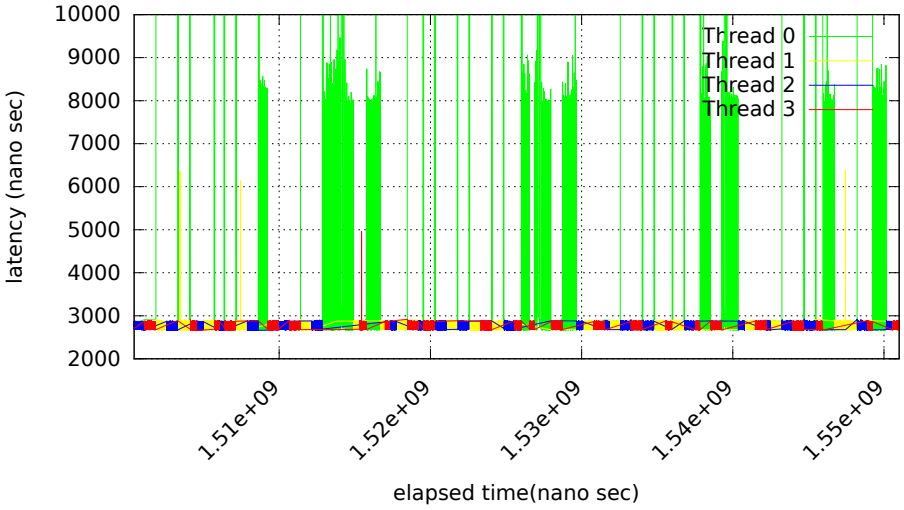

Fig. 6. Execution overhead for multiple threads pinned to a single core, with page fault events occurring.

To induce a high memory footprint we use the same technique as in Section 3.2, where we create an enclave which exceeds in size the amount of available

physical PRM. We expect some performance degradation for multiple threads running on the same logical core executing within the enclave. When a page fault occurs, all threads running on the particular core must exit the enclave and block until the page fault is serviced. Our experiment therefore consists of two modes, one where we pin all threads to separate logical cores, and one where we pin all threads to a single core. Both experiments dedicate a single thread to interrogating every 4 kB page of the heap memory causing regular page faults to occur. Our test bench has 4 logical cores so our experiment runs a total of 4 threads simultaneously for both experiments. The remaining threads are just busy-waiting in a loop, measuring the time taken in each iteration. Figure 5 illustrates 4 threads pinned to 4 different cores where core 0 is interrogating memory and causing page faults to occur as illustrated in the green spikes. We observe that there is no co-dependency between threads, and the 3 remaining threads are not impacted by interrupts occurring on the former.

Our second experiment demonstrates the opposite. We force all 4 threads to be pinned to a single logical core, and as a consequence we observe that thread 0, who is causing interrupts to occur, is blocking all other threads from executing while servicing the costly page faults. It is worth noting that this is how threads behave in regular process address space when faced with a hardware interrupt. However, page faults are more costly to perform in enclave memory and more frequent as previous experiments show due to memory footprint constraints. Secondly, we observe that thread scheduling behaves differently as well. Context switches between threads executing on the same logical core happens multiple magnitudes more infrequently than regular threads executing outside of enclaves. We theorize that this is a design choice when implementing enclave support, because interrupts in enclaves are especially costly. Any context switch would have to be induced by the timer hardware interrupt triggering the thread to exit the enclave, and so it makes sense increasing the scheduler time slices to amortize this cost.

4 Discussion

From the micro benchmarks detailed in Section 3, we pinpoint several performance traits of SGX that should be taken into consideration when designing trusted computing-enabled cloud services. We classify these individually as the cost of entering and exiting enclaves, the cost of data copying, the cost of provisioning new enclaves, the cost of memory usage and the cost of multithreaded execution.

Section 2 explained that the transitioning cost is uniform in terms of cost with respect to direction. Moreover, the most significant cost is attributed to the buffer size input as argument to the transition. More specifically, from Figure 2 we observe a sharp rise in cost when buffer sizes are larger than 64 kB. We conjecture that this is an architectural boundary, where encalves are pre-provisoned, by default, with a given number of pages. Future iterations of SGX may alter this

behavior, opting for an increase in pre-provisioned pages. Our principles therefore state:

The Size Principle: *The size of an enclave should not exceed the architecturaly determined pre-provisoned memory resources.*

The Cohesion Principle: *Applications should partition its functional components to minimize data copied across enclave boundaries.*

Following the latter principle, a possible component architecture would be to co-locate all application logic into a single, self-sufficient enclave. Haven [3], is a prominent example of this approach. By means of a library OS, a large part of the system software stack is placed within a single enclave, reducing the interface between trusted and untrusted code. However, this approach directly contradicts the observation made in Section 3.2 regarding the cost of having a large memory footprint. Since the EPC is a limited resource, the SGX page fault handler promptly pages out enclave memory not being used. However, the paging experiment demonstrates that the available pool of EPC memory is not exhausted, even in the presence of high memory contention. As detailed in Section 3.2, the faulting address is not provided as part of the page fault event and the page fault handler is therefore not able to make any assumptions about the memory access patterns. We therefore state that:

The Access Pattern Principle: *Prior knowledge about application's memory consumption and access pattern should be used to modify the SGX kernel module in order to reduce memory page eviction.*

Our experiments have demonstrated that enclave creation is costly in terms of provisioning latency. By pre-provisioning enclaves whenever usage patterns can be predicted, the application is able to hide some of this cost. However, once used, an enclave might be tainted with secret data. Recycling used enclaves to a common pool can therefore potentially leek secrets from one domain to the next; invalidating the isolation guarantees. We therefore state that:

The Pre-provisioning Principle: *Application authors that can accurately predict before-the-fact usage of enclaves should pre-provision enclaves in a disposable pool of resources that guarantees no reuse between isolation domains.*

The cost of enclave creation must also factor in the added baseline cost of storing metadata structures associated with each enclave in memory. Provisioning enclaves must at least account for its SECS, one TCS structure for each logical core executed inside an enclave, and one SSA for storing secure execution context for each thread. [5] details that to simplify implementation, most of these structures are allocated at the beginning of an EPC page wholly dedicated to

that instance. Therefore, enclaves executing on 4 logical cores may have 9 pages (34 kB) in total allocated to it, excluding code and data segments. Applications should consider the added memory cost of separate enclaves in conjunction with the relative amount of available EPC. Furthermore, to offset the cost of having multiple enclaves, application authors should consider security separation at a continuous scale. Some security models might be content with role based isolation, rather than call for an explicit isolation of all users individually. We therefore state that:

The Isolation Principle: *Application authors should carefully consider the granularity of isolation required for their intended use, as a finer granularity includes the added cost of enclave creation.*

Executing multiple threads from the same core inside a single enclave degrades the concurrent performance by blocking execution when servicing a page fault. Although regular non-enclave execution behaves similarly, the overhead associated with enclave page faults becomes significant when memory footprint increases. Moreover, latency critical applications will suffer because of the increased time slices of thread interrupts initially thought to amortize the cost of exiting enclaves when switching contexts. From this we deduce that the number of threads executing inside enclaves should never exceed the logical core count for a given system. We therefore establish the following principle:

The Affinity Principle: *Applications should not affinitize multiple threads to the same core.*

Section 3.1 demonstrates the cost of transitioning into and out of an enclave, and it becomes evident that to reduce the transitioning overhead threads should be pinned inside enclaves. Enclave threads should rather transport data out of the enclave through writing to regular DRAM and similarly poll for incoming data. We therefore state:

The Pinning Principle: *Application authors should pin threads to enclaves to avoid costly transitions.*

The prior statements lead us to the following principle:

The Asynchrony Principle: *All execution inside enclaves should be asynchronous.*

Threads should be pinned inside enclaves to amortize transition cost and total thread count should not exceed logical core count. Application authors must therefore be diligent in terms of assigning threads to enclaves. Applications might further isolate contexts based on either user or tenant in different

mutually distrusting enclaves, each of which requires a dedicated thread. Core logic executing inside enclaves should remain responsive at all time, servicing both incoming requests and processing data. We therefore state that rather than allocating multiple threads to the same enclave, all execution should be fully asynchronous. This furthermore has the added benefit of high resource utilization improving overall application performance.

At the time of writing, the only available hardware supporting SGX are the Skylake generation Core chips with SGX version 1. Our experiments demonstrate that paging has a profound impact on performance and a natural follow-up would be to measure the performance characteristics of dynamic paging support proposed in the SGX version 2 specifications.

SGX supports attestation of software running on top of an untrusted platform, by using signed hardware measurements to establish trust between parties. For future efforts it would be interesting, in light of the large cost of enclave transition demonstrated above, to examine the performance characteristics of a secure channel for communication between enclaves.

5 Related Work

Several previous works quantify various aspects of the overhead associated with composite architectures based on SGX. Haven [3] implements shielded execution of unmodified legacy applications by inserting a library OS entirely inside of SGX enclaves. This effort resulted in architectural changes to the SGX specification to include, among other things, support for dynamic paging. The proof-of-concept implementation of Haven is only evaluated in terms of legacy applications running on top of the system. Furthermore, Haven was built on a pre-release emulated version of SGX, and the performance evaluation is not directly comparable to real world applications. Overshadow [4] provides similar capabilities as Haven, but does not rely on dedicated hardware support.

SCONE [2] implements support for secure containers inside of SGX enclaves. The design of SCONE is driven by experiments on container designs pertaining to the TCB size inside enclaves, in which, at the most extreme an entire library OS is included and at the minimum a stub interface to application libraries. The evaluation of SCONE is much like the evaluation of Haven, based on running legacy applications inside SCONE containers. While Arnautov et al. [2] make the same conclusions with regards to TCB size versus memory usage and enclave transition cost as Baumann et al. [3], the paper does not quantify this cost. Despite this, SCONE supplies a solution to the entry-exit problem we outline in Section 3, where threads are pinned inside the enclave, and do not transition to the outside. Rather, communication happens by means of the enclave threads writing to a dedicated queue residing in regular DRAM memory. This approach is still, however, vulnerable to theads being evicted from enclaves by AEX caused by an Inter Processor Interrupt (IPI) as part of a page fault.

Costan and Devadas [5] describe the architecture of SGX based on prior art, released developer manuals, and patents. Furthermore, they conduct a compre-

hensive security analysis of SGX, falsifying some of its guarantees by explaining in detail exploitable vulnerabilities within the architecture. This work is mostly orthogonal to our efforts, yet we base most of our knowledge of SGX from this treatment on the topic. These prior efforts lead Costan et al. [6] to implement Sanctum, an alternative hardware architectural extension providing many of the same properties as SGX, but targeted towards the Rocket RISC-V chip architecture. This paper evaluates its prototype by simulated hardware, against an insecure baseline without the proposed security properties. McKeen et al. [11] introduce dynamic paging support to the SGX specifications. This prototype hardware was not available to us.

Ryoan [9] attempts to solve the same problems outlined in the introduction, by implementing a distributed sandbox facilitating untrusted computing on secret data residing on third-party cloud services. Ryoan proposes a new request oriented data-model where processing modules are activated once without persisting data input to them. Furthermore, by remote attestation, Ryoan is able to verify the integrity of sandbox instances and protect execution. By combining sandboxing techniques with SGX, Ryoan is able to create a shielding construct supporting mutually distrust between the application and the infrastructure. Again, Ryoan is benchmarked against real world applications, and just like other prior work, does not correctly quantify the exact overhead attributed to SGX primitives. Furthermore, large parts of its evaluation is conducted in an SGX emulator based on QEMU, which has been retrofitted with delays and TLB flushes based upon real hardware measurements to better mirror real SGX performance. These hardware measurements are present for EENTRY and EEXIT instructions, but do not attribute the cost of moving argument data into and out of enclave memory. Moreover, Ryoan speculates on the cost of SGX V2 paging support, although strictly based on emulated measurements and assumptions about physical cost.

ARM TrustZone is a hardware security architecture that can be incorporated into ARMv7-A, ARMv8-A and ARMv8-M on-chip systems [12, 15]. Although the underlying hardware design, features, and interfaces differ substantially to SGX, both essentially provide the same key concepts of hardware isolated execution domains and the ability to bootstrap attested software stacks into those enclaves [14]. However, the TrustZone hardware can only distinguish between two execution domains, and relies on having a software based trusted execution environment for any further refinements.

6 Conclusion

SaaS providers are increasingly storing personal privacy-sensitive data about customers on third-party cloud providers. Moreover, companies monetize this data by providing personalized experiences for customers requiring curation and analysis. This dilution of responsibility and trust is concerning for data owners as cloud providers cannot be trusted to enforce the, often government mandated, restrictive usage policies which accompany privacy-sensitive data.

Intel SGX is part of a new wave of trusted computing targeting commodity hardware and allowing for the execution of code and data in trusted segments of memory at close to native processor speed. These extensions to the x86 ISA guarantee confidentiality, integrity and correctness of code and data residing on untrusted third-party platforms.

Prior work demonstrates the applicability of SGX for complete systems capable of hosting large legacy applications. These systems, however, do not quantify the exact micro architectural cost of achieving confidentiality, integrity and attestation for applications through the use of trusted computing. This paper has evaluated the cost of provisioning, data copying, context transitioning, memory footprint and multi-threaded execution of enclaves. From these results we have distilled a set of principles which developers of trusted analytics systems should use to maximize the performance of their application while securing privacy-sensitive data on third-party cloud platforms.

Acknowledgments

This work was supported in part by the Norwegian Research Council project numbers 263248/O70 and 250138. We would like to thank Robbert van Renesse for his insights and discussions, and anonymous reviewers for their useful insights and comments.

References

[1] 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*. Vol. 13. 2013.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 689–703.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding applications from an untrusted cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Advanced Computing Systems Association, 2014.

[4] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. "Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems". In: *13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: ACM, 2008, pp. 2–13. DOI: 10.1145/1346281.1346284.

[5] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *Cryptology ePrint Archive*. 2016.

[6] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal hardware extensions for strong software isolation". In: *USENIX Security*. Vol. 16. 2016, pp. 857–874.

[7] Anders T. Gjerdrum, Håvard D. Johansen, and Dag Johansen. "Implementing Informed Consent as Information-Flow Policies for Secure Analytics on eHealth Data: Principles and Practices". In: *IEEE Conference on Connected Health: Applications, Systems and Engineering Technologies: The 1st International Workshop on Security, Privacy, and Trustworthiness in Medical Cyber-Physical System*. CHASE '16. IEEE, June 2016. DOI: http://dx.doi.org/10.1109/CHASE.2016.39.

[8] Anders T. Gjerdrum, Robert Pettersen, Håvard D. Johansen, and Dag Johansen. "Performance of Trusted Computing in Cloud Infrastructures with Intel SGX". In: *7th International Conference on Cloud Computing and Services Science*. CLOSER '17. SCITEPRESS, 2017.

[9] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data". In: *12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 533–549.

[10] Håvard D Johansen, Eleanor Birrell, Robbert Van Renesse, Fred B Schneider, Magnus Stenhaug, and Dag Johansen. "Enforcing Privacy Policies with Meta-Code". In: *6th Asia-Pacific Workshop on Systems*. ACM. 2015, p. 16.

[11] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. "Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave". In: *Hardware and Architectural Support for Security and Privacy 2016*. ACM. 2016, p. 10.

[12] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. "TrustZone Explained: Architectural Features and Use Cases". In: *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*. IEEE. 2016, pp. 445–451.

[13] Justin D. Osborn and David C. Challener. "Trusted Platform Module Evolution". In: *Johns Hopkins APL Technical Digest* 32.2 (2013), pp. 536–543.

[14] Robert Pettersen, Håvard D. Johansen, and Dag Johansen. "Trusted Execution on ARM TrustZone". In: *7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. Apr. 2017.

[15] Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta Ur Rehman Khan, Sajjad A Madani, Samee U Khan, and Albert Y Zomaya. "A survey of mobile device virtualization: taxonomy and state of the art". In: *ACM Computing Surveys (CSUR)* 49.1 (2016), p. 1.

[16] TCG Published. *TPM Main Part 1 Design Principles*. Specification Version 1.2 Revision 116. Trusted Computing Group, Mar. 2011.