Performance of Trusted Computing in Cloud Infrastructures with Intel SGX

Anders T. Gjerdrum, Robert Pettersen, Håvard D. Johansen and Dag Johansen
UiT: The Arctic University of Norway, Tromsø, Norway

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

Abstract: Sensitive personal data is to an increasing degree hosted on third-party cloud providers. This generates strong concerns about data security and privacy as the trusted computing base is expanded to include hardware components not under the direct supervision of the administrative entity responsible for the data. Fortunately, major hardware manufacturers now include mechanisms promoting secure remote execution. This paper studies Intel's Software Guard eXtensions (SGX), and experimentally quantifies how basic usage of this instruction set extension will affect how cloud hosted services must be constructed. Our experiments show that correct partitioning of a service's functional components will be critical for performance.

1 INTRODUCTION

Sensors and mobile devices record ever more aspects of our daily lives. This is causing an influx of data streams that feeds into potentially complex analytical pipelines hosted remotely by various cloud providers. Not only are the sheer amounts of data generated cumbersome to store and analyze at scale; data might also be accompanied by strict privacy requirements, as is the case with smart home and health monitoring devices (Gjerdrum et al., 2016).

Processing of sensitive and personal data in the cloud requires the design of new Software-as-a-Service (SaaS) architectures that are able to enforce rigid privacy and security policies (Johansen et al., 2015) throughout the entire hardware and software stack, including the underlying cloud-provided Infrastructure-as-a-Service (IaaS) components. Although, commodity hardware mechanisms for trusted computing have been available for some time (TCG Published, 2011; Osborn and Challener, 2013), these are often poised with performance and functionality restrictions. Prior implementations by Intel, like Trusted Platform Modules (TPM) and Trusted Execution Technology (TXT), are able to establish trust and guarantee integrity of software, the latter also supporting rudimentary secure code execution.

Software Guard Extentions (SGX) (Anati et al., 2013) is Intel's new trusted computing platform that, together with similar efforts by both ARM and AMD, is quickly making general trusted computing a commodity. Fundamentally, SGX is an instruction set extension introduced with the Skylake generation of Intel's Core architecture, supporting confidentiality, integrity and attestation of trusted code running on untrusted platforms. SGX is able to counter a multitude of different software and physical attacks by the construction of secure enclaves consisting of trusted code and data segments. While SGX should be considered an iterative technology built on previous efforts, it surpasses previous iterations both in terms of performance and functionality. SGX is designed to provide general secure computing facilities allowing developers to easily port their existing legacy applications into SGX enabled enclaves. These properties make SGX an attractive technology for cloud-based SaaS architectures that handle person sensitive data.

SGX is a proprietary platform and prior knowledge is based on limited documentation describing its architecture. Furthermore, little is known about the performance of the primitives provided by the SGX platform and how to author software utilizing these primitives while maximizing performance.

In this paper we analyze the performance characteristics of the SGX technology currently available to better understand how such technologies can be used to enforce privacy policies in cloud hosted SaaS architectures. We analyze SGX primitives at a fine-grained level and provide detailed performance evaluation of the core mechanisms in SGX. The paper is structured as follows: Section 2 outlines the relevant parts of the SGX micro architecture while Section 3 outlines the

Gjerdrum, A., Pettersen, R., Johansen, H. and Johansen, D.
Performance of Trusted Computing in Cloud Infrastructures with Intel SGX.
DOI: 10.5220/0006373706960703
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 668-675
ISBN: 978-989-758-243-1

details of our microbenchmark. Section 4 provides an informed discussion of our findings and Section 5 detail relevant work before concluding remarks.

2 INTEL SOFTWARE GUARD EXTENTIONS (SGX)

SGX allows regular application threads to transition into secure enclaves by issuing the special EENTER special instructions to a logical processor. Entry is initiated by performing a controlled jump into the enclave code, analogous to how entry into virtual machine contexts occurs. A process can only enter an enclave from ring 3, i.e user level, and threads running in *enclave mode* are not allowed to trigger software interrupts, also prohibiting the use of system calls. An application which requires access to common Operating System (OS) provided services, like the file system, must be carefully designed so that its threads exit *enclave mode* through application defined interfaces before invoking any system calls. Since SGX's Trusted Computing Base (TCB) does not include the underlying OS, all such transitions, parameters, and responses, must be carefully validated by the application designer.

SGX allows multiple threads to execute inside the same enclave. For each logical processor executing inside an enclave a Thread Control Structure (TCS) is needed. These data structures must be provisioned before enclave startup, and are stored in the Enclave Page Cache (EPC) main-memory pages set aside for enclaves. Among other things, the TCS contains the OENTRY field which is loaded into the instruction pointer when entering an enclave. Before doing so, SGX stores the execution context of the untrusted code into regular memory, by using the XSAVE instruction, which then again is restored when exiting the enclave. Stack pointers are not modified when entering an enclave, however (Costan and Devadas, 2016) suggests that to avoid the possibility of exploits, it is expected that each enclave set their stack pointer to an area fully contained withing EPC memory. Parameter input to the enclave is marshalled into buffers, and once the transition is done, enclave code can copy data directly from untrusted DRAM memory. This is not part of the native SGX implementation, rather a convenience provided by the application SDK.

Threads exit enclaves either voluntarily through synchronous exit instructions, or asynchronously by service of a hardware interrupt occurring on the affected logical core. Synchronous exits, through the EEXIT instruction, causes the logical processor to leave enclave mode. The instruction pointer as well as the stack pointers are restored to their prior address before entering the enclave. SGX does not modify any instructions on enclave exit and so it is the authors' responsibility to clear them, to avoid leaking secret information. In the case of an Asynchronous Enclave Exit (AEX), a hardware interrupt such as a page fault causes the processor to exit the enclave and jump down to the kernel in order to service the fault. Prior to this, SGX saves the execution context into EPC memory for safekeeping, before clearing it so that the OS is not able to infer any execution state from the enclave. When the interrupt handler is done, SGX restores the execution context and resumes execution.

2.1 The Enclave Page Cache

Memory used by enclaves is separated at boot time from regular process DRAM memory into what is called Processor Reserved Memory (PRM). This contiguous region of memory is divided into 4 kb pages, collectively referred to as the Enclave Page Cache (EPC). EPC memory is only accessible inside the enclave or via the SGX instruction set. Neither system software running at protection ring 0 (kernel mode) or application code at ring 3 (user mode) are able to access its memory contents directly, and any attempt to read or write to it is ignored. Furthermore, DMA access to PRM memory is prohibited by hardware to guard against malicious peripheral devices attempting to tap the system bus. The confidentiality of the enclave is guarded by Intel's Memory Encryption Engine (MEE), which encrypts and decrypts memory at the CPU package boundary, on the system bus right after the L3 cache.

Similar to virtual memory, EPC page management is handled entirely by the OS. However, EPC memory is not directly accessible to any system mode and each page assignment is done through SGX instructions. The OS is responsible for assigning pages to particular enclaves and evict pages to regular DRAM. The current generation of SGX hardware only supports a maximum PRM size of 128 MB, but through swapping, there are no practical limits to the size of an enclave. The integrity of pages swapped out is guaranteed by always checking an auxiliary data structure also residing in PRM, called the Enclave Page Cache Map. This datastructure contains the correct mappings between virtual addresses and Physical PRM memory, as well as integrity checks for each page. Each page can only belong to one enclave, and as a consequence, shared memory between enclaves is prohibited. They are however able to share DRAM memory if residing inside the same process'

address space, and enclave memory is allowed to read and write to untrusted memory inside that process. The page eviction instruction also generates a liveness challenge for each page, storing them in special EPC pages for later comparison. 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 order to guard against stale address translations for executing enclaves, the processor does a coarse-grained TLB shootdown for pages being evicted. Page faults targeting a particular enclave will cause the kernel to issue a Inter Processor Interrupt (IPI) for all logical cores running inside of the enclaves in question. This will cause each thread to do an AEX, as mentioned above, and trap down to the kernel page fault handler. Moreover, the lowermost 12 bits of the virtual address at fault, stored in the CR2 registry, is cleared so that the OS cannot infer any access pattern. To amortize the cost of interrupting all cores executing inside a particular enclave for each page eviction, the SGX implementation supports batching up to 16 page evictions together at a time.

2.2 Enclave Creation

SGX supports multiple mutually distrusting enclaves on a single machine either within the same process' address space or in different processes. Enclaves are created by system software on behalf of an application, issuing an ECREATE instruction. This will cause SGX to allocate a new EPC page for the SGX Enclave Control Structure (SECS) which stores metadata for each enclave. This is used by SGX instructions to identify enclaves, and among other things map enclaves to physical EPC pages via the EPCM structure. Before the enclave is ready for executing code, each initial code and data segment must be added to enclave memory via the OS issuing specially crafted instructions to the SGX implementation for each page. The same instruction is also used to create the TCS for each expected thread inside the enclave. In addition, the OS driver issues updates for enclave measurements used for software attestation. We refer to the SGX developer manual for a description of the SGX attestation process. When all pages are loaded, the enclave is initialized and the enclave receives a launch token from a special pre-provisioned enclave entrusted by Intel. At this point, the enclave is considered fully initialized and no further memory allocations may happen. Intels revised specifications for SGX version 2 includes support for expanding enclaves after initial creation by dynamic paging support. However, we refrain from further explanation as hardware supporting these specifications has not been released at this point.

When an enclave is destroyed, the inverse happens, as the OS marks each page used by the enclave as invalid by the EREMOVE instruction. Before freeing the page, SGX makes sure that no logical processor is executing inside the enclave that owns the particular page. Finally, the SECS is deallocated if all pages in the EPCM referring to that particular enclave are deallocated.

3 EXPERIMENTS

To gain experience in how the next generation cloud-based SaaS systems should be architected to best take advantage of the SGX features in modern processors, we ran a series of micro benchmarks on SGX-enabled 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. To avoid inaccuracies caused by dynamic frequency scaling, Intel Speedstep and CStates were disabled in all our experiments. To measure the peak performance of the architecture, we also altered the PRM size in hardware setup to be the maximum allowed 128 MB. We run the experiments on Ubuntu 14.04 using the open source kernel module for Intel SGX.[1] We instrumented the SGX kernel module to record the operational costs. Based on our understanding of the system we derived different benchmarks testing various features of the platform. Common for all experiments is the observation that more iterations did not yield a lower deviation. This may be attributed to noise generated by the rest of the system. This noise is subtle, but significant since we are measuring at fine-grained time intervals.

Note that the current iteration of SGX prohibits use of the RDTSC instruction inside of enclaves, and as such there are no natively timing facilities available inside enclaves. A later release reveals that the updated specifications for SGX version one does support RDTSC inside enclaves. Hints suggests that this might be distributable by means of an update to the microcode architecture. We were, however, unsuccessful in obtaining this update. Time measurements performed throughout this experiment must therefore exit the enclave before being captured. As a consequence, we can only measure the total time taken between entering and exiting an enclave described as the sequence of events depicted in Figure 1.

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

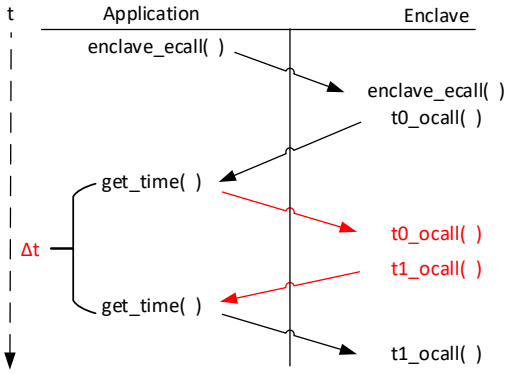

Figure 1: Sequence of events involved in measuring time spent inside enclaves.

3.1 Entry and Exit Costs

In our first experiments, we look at the cost of entering and leaving an enclave. Understanding this cost is important as it dictates how SGX enabled SaaS services can partition its functionality between enclaved and non-enclaved execution to minimize TCB size. A prohibitively large cost of entry would necessitate a reduction in the number of entry calls, and thus increasing the amount of code and data residing inside of the enclave, increasing the required TCB. The extreme case being a full library OS that include almost all the functionality an application requires within the enclave (Baumann et al., 2014). The Intel Software Developer Manual[2] suggests that the cost of entering an enclave is also a function of the size of the data copied into the enclave as a part of the entry. Thus, if experiments show that the cost of large amounts of data entering the enclave is prohibitively large, only data requiring confidentiality should be copied into the enclave.

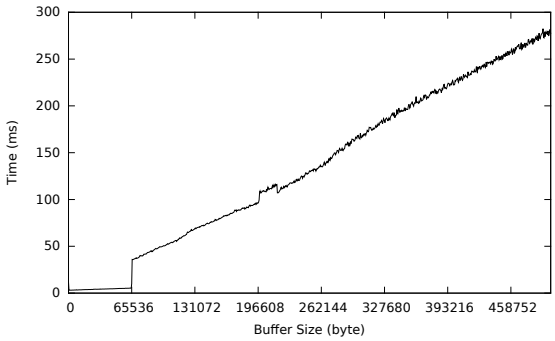

Figure 2: Enclave transition cost as a function of buffer size.

Figure 2 shows the measured cost as a function of increasing buffer sizes. As shown in the figure, the cost of transitioning into enclaves increases linearly with the buffer size. This experiment only uses buffers as parameter while transitioning into the enclave. To be able to host the buffer inside the enclave, its heap size must be sufficiently large. The observed baseline cost with no buffer is the bare transition cost for entering enclaves. This cost quickly becomes insignificant as the buffer size increases. This behavior is expected as this cost includes copying the buffers into enclave memory on transitions, which invokes the MEE for memory written to the enclave. To our surprise, however, we observed that the baseline cost only increased above 64 kb. One possible explanation for this is that the pages may already be present in EPC memory for buffer sizes smaller than 64 kb

For larger buffers the increased cost can also be attributed to page faults caused by enclave memory previously evicted to DRAM. This issue is further explored in the next experiment.

3.2 Paging

Another probable architectural trade-off is the logical assumption that an increase in TCB would reduce enclave transitions but requires more PRM. As mentioned in Section 1, the PRM is a very limited resource in comparison to regular DRAM and the system has a total of 128 MB of it. Moreover, any enclave is subject to the system software evicting EPC pages when PRM resources becomes scarce. Any system using SGX should factor in the cost of swapping pages between PRM and regular DRAM. Figure 3 illustrates this cost in enclaves as observed by both the kernel and the user level enclave.

The y-axis is the discrete cost in nano seconds, while the x-axis is time elapsed into the experiment. We instrumented the OS kernel driver to measure the time taken to evict pages out of EPC into DRAM denoted by red dots, as well as the total time spent inside the page fault handler, shown by the black line.

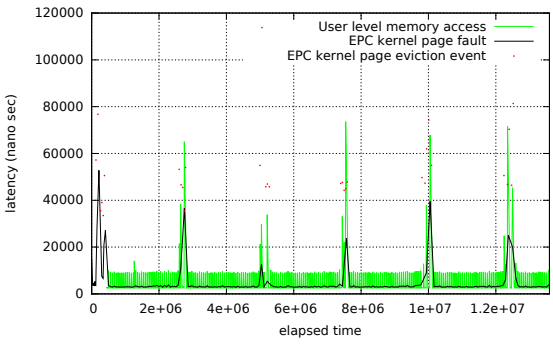

Figure 3: Paging overhead in nano seconds as a function of time elapsed while writing sequentially to enclave memory.

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

The green line denotes user level instrumentation measuring the time it takes to write to a particular address in EPC memory. Similarly to the prior experiment, we are prohibited to make timing measurements inside enclaves. Therefore, the user level measurements include the baseline cost of entry and exit of an enclave, notably with 4 byte buffers transitioning each way.

To induce enclave page faults we set the total enclave heap size to 256 MB, which is larger than the total amount of available EPC memory. Furthermore, to hit each page we invoke write operations to each address within the 4 kb page size sequentially along the allocated memory space inside the enclave. As mentioned in Section 2, the only time enclaves are able to allocate memory is before the EINIT instruction is called by issuing EADD. Therefore, all memory must be allocated before enclave execution begins. We can clearly see at the beginning of the experiment an increase in page faults occurring when trying to fit 256 MB of enclave memory into potentially 128 MB of physical EPC memory.

Correlating the different events happening at user level and kernel level we observe a strong relationship between eviction events and increase in write time at user level. One property of the system that might increase this cost is the fact that evicting pages causes AEX events for any logical processor executing within an enclave, as explained in Section 2.

We also observe that the kernel driver is operating very conservatively in terms of assigning EPC pages to enclaves by the amount of page faults occurring during execution. Moreover, as mentioned in Section 2, the 12 lower bits of the virtual page fault address is cleared by SGX before trapping down to the page fault handler. Therefore, the driver is not able to make any assumptions about memory access patterns inside enclaves. Moreover, as Section 2 explains, liveness challenge data might also be evicted of EPC memory, causing a cascade of page loads to occur from DRAM. It is worth noting that our experiment only uses one thread, and that all page evictions issuing IPI only interrupt this single thread.

It is clear that high performance applications might want to tune the OS support for paging to their needs. If an application can predict a specific access pattern, the kernel paging support should adapt to this. Moreover, by optimizing towards exhaustive use of the EPC memory, applications running inside enclaves might be subject to fewer page faults.

Furthermore, initial setup will keep large amounts of the enclave in memory, which might eliminate the overhead of paging for some enclaves. This furthermore reduces overhead caused by IPI interrupts triggering AEX from the given enclave. Initially, the creation of large enclaves trigger memory allocations by the kernel, and it might be necessary for application developers to offset this initial cost by previsioning enclaves.

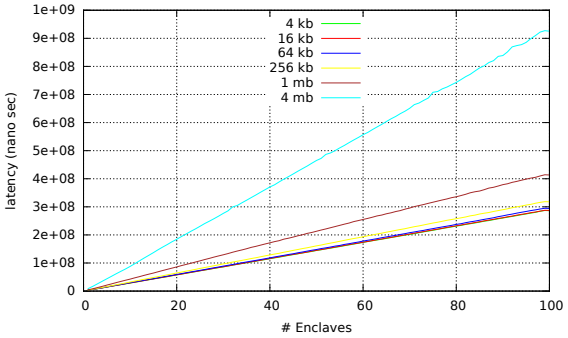

Figure 4: Latency as a function of number of enclaves created simultaneously, for differing sizes of enclaves

3.3 Enclave Provisioning

Modern distributed system architectures increasingly rely on modular programing paradigms and multi-component software with possibly differing trust domains. Such distributed systems often consist of several third-party open source components, both trusted and not. Moreover, separating both the unit of failure and trust of such systems is often a good idea.

SGX supports the creation of multiple mutually distrusting enclaves which can be used in such a modular design. As mentioned in Section 2 the SGX programming model allows enclaves to communicate with the outside using well defined interfaces, which lends itself to an architecture where trust is compartmentalized into separate enclaves. Figure 4 illustrates the additional cost in terms of provisioning latency as a function of enclaves created simultaneously, and we can clearly observe that the added cost in enclave creation increases linearly. Through multiple iterations of this experiment we observe the added cost by increasing enclave sizes. As demonstrated, this added cost becomes increasingly significant when provisioning multiple enclaves exceeding 256 kb in size. As mentioned in Section 2, enclaves are created by issuing specially crafted functions for each page of code and data being allocated inside enclave memory. It is worth mentioning that we observed a significant amount of page faults occurring during enclave creation, and it is reasonable to assume that this is also contributing to the cost. Furthermore, the observations made about entry cost for buffer sizes less than 64 kb shown in Figure 2, is further corroborated by the fact that for enclave sizes less than 64 kb the provisioning costs are nearly identical.

For application software requiring low latency operation it might be necessary to pre-provision enclaves to offset this cost in latency. However, this approach might cause additional problems with collocating them in EPC memory if the individual enclaves are sufficiently large.

4 DISCUSSION

From our experiments in Section 3, we have identified several important performance idiosyncrasies of SGX that should be considered when constructing SGX enabled cloud services: the cost of entering and exiting enclaves, the cost of data copying, the cost of provisioning new enclaves and the cost of memory usage.

As mentioned in Section 2, entry and exit procedures do similar amounts of work in terms of cost. As our experiments show, the most significant cost factor of transitioning is the buffer size input as argument to the transition either through entry or exit. In particular, we observed a steep rise in data copy cost when buffer sizes are larger than 64 kb. Our recommendation is therefore that:

Recommendation 1. *Applications should partition its functional components to minimize data copied across enclave boundries.*

One possible component architecture that follows the guideline of Recommendation 1 would be to co-locate all functionality into one single enclave, making it largely self-sufficient. An example a system following such an approach is Haven (Baumann et al., 2014), which reduces the interface between trusted and untrusted code by co-locating a larger part of the system software stack inside a single enclave by means of a library OS. The efficiency of this approach, however, directly contradicts the observation we made in Section 3.2, where we measured the overhead associated with enclave memory being paged in and out to regular DRAM. Because the EPC is a scarce resource, system software aggressively pages out enclave memory not being used. However, as our experiments show, the page fault handler is overeager, and fails to fully utilize EPC memory exhaustively. Because of security concerns, the kernel is not given the exact faulting address of each enclave page fault, and therefore does not make any assumptions as to the memory access patterns. We therefore recommend that:

Recommendation 2. *The size of an enclave should not exceed 64 kb.*

Recommendation 3. *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.*

As our experiment shows, enclave creation is costly and time consuming. To hide some of this cost, the underlying OS can pre-provision enclaves whenever usage patterns can be predicted. However, once used, an enclave might be tainted with secret data. Recycling used enclaves to a common pool can therefore potentially leeks secrets from one process to the next: invalidating the isolation guarantees. We therefore recommend that:

Recommendation 4. *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 metadata structures associated with each enclave. Provisioning an enclaves must at least account for its SECS, one TCS structure for each logical core executed inside an enclave, and one SSA for each thread performing AEX. (Costan and Devadas, 2016) explains that to simplify implementation, most of these structures are allocated at the beginning of a EPC page, wholly dedicated to that instance. Therefore, it is not out of line to consider an enclave with 4 logical cores, having 9 pages (34 kb) 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 recommend that:

Recommendation 5. *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.*

At the time of writing, the only available hardware supporting SGX are the Skylake generation Core chips with SGX version 1. As our experiments show, paging has a profound impact on performance, and a natural follow-up would be to measure the performance characteristics of the dynamic paging support proposed in the SGX V2 specifications. However,

as mentioned earlier, Intel has yet to release any information regarding the arrival of SGX V2 enabled chips. The imminent 8th generation Kaby Lake chips do not include support, and the earliest likely release will therefore as part of Cannon Lake in Q4 2017.

SGX supports attestation of software running on top of untrusted platforms, by using signed hardware measurements to establish trust between parties. These parties could be either locally with two distinct enclaves executing on the same hardware, or remotely by help of Intel's attestation service. In the future, 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 quantifies various aspects of the overhead associated with composite architectures based on SGX. Haven (Baumann et al., 2014) 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 (Chen et al., 2008) provide similar capabilities as Haven, but does not rely on dedicated hardware support.

SCONE (Arnautov et al., 2016) 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., 2016) make the same conclusions with regards to TCB size versus memory usage and enclave transition cost as (Baumann et al., 2014), they do 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 enclaves, 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 IPI as part of a page fault.

(Costan and Devadas, 2016) describe the architecture of SGX based on prior art, released developer manuals, and patents. Furthermore, they conduct a comprehensive 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, however, we base most of our knowledge of SGX from this treatment on the topic. These prior efforts lead (Costan et al., 2016) to implement Sanctum, which implements an alternative hardware architectural extension providing many of the same properties as SGX, but targeted towards the Rocket RISC-V chip architecture. Sanctum evaluates its prototype by simulated hardware, against an insecure baseline without the proposed security properties. (McKeen et al., 2016) introduce dynamic paging support to the SGX specifications. This prototype hardware were not available to us.

Ryoan (Hunt et al., 2016) attempts to solve the same problems outlined in the introduction, by implementing a distributed sandbox for facilitating untrusted computation 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 have 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, however do not attribute the cost of moving argument data into and out of enclave memory. Moreover, Ryoan speculate 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 (Ngabonziza et al., 2016; Shuja et al., 2016). 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. 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

Online services are increasingly relying on third-party cloud providers to host sensitive data. This tendency brings forth strong concerns for the security and privacy of data owners as cloud providers cannot fully be trusted to enforce the restrictive usage policies that often govern such data. Intel SGX provides hardware support for general trusted computing in commodity hardware. These extensions to the x86 instruction set establish trust through remote attestation of code and data segments provisioned on non-trusted infrastructure, furthermore guaranteeing the confidentiality and integrity of these from potentially malicious system software.

Prior efforts demonstrate the capabilities of SGX through rigorous systems capable of hosting large legacy applications securely inside enclaves. These systems, however, do not quantify the exact cost associated with using SGX. This paper evaluates the micro architectural cost of entering and exiting enclaves, the cost of data copying, the cost of provisioning new enclaves and the cost of memory usage. From this, we have derived five recommendations for application authors wishing to secure their cloud-hosted privacy sensitive data using SGX.

ACKNOWLEDGMENTS

This work was supported in part by the Norwegian Research Council project numbers 231687/F20. We would like to thank the anonymous reviewers for their useful insights and comments.

REFERENCES

Anati, I., Gueron, S., Johnson, S., and Scarlata, V. (2013). Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13.

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. (2016). Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, GA. USENIX Association.

Baumann, A., Peinado, M., and Hunt, G. (2014). 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.

Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. (2008). Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 2–13, New York, NY, USA. ACM.

Costan, V. and Devadas, S. (2016). Intel sgx explained. In *Cryptology ePrint Archive*.

Costan, V., Lebedev, I., and Devadas, S. (2016). Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, volume 16, pages 857–874.

Gjerdrum, A. T., Johansen, H. D., and Johansen, D. (2016). Implementing informed consent as information-flow policies for secure analytics on eHealth data: Principles and practices. In *Proc. of the 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.

Hunt, T., Zhu, Z., Xu, Y., Peter, S., and Witchel, E. (2016). Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 533–549, Berkeley, CA, USA. USENIX Association.

Johansen, H. D., Birrell, E., Van Renesse, R., Schneider, F. B., Stenhaug, M., and Johansen, D. (2015). Enforcing privacy policies with meta-code. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 16. ACM.

McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., and Rozas, C. (2016). Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 10. ACM.

Ngabonziza, B., Martin, D., Bailey, A., Cho, H., and Martin, S. (2016). Trustzone explained: Architectural features and use cases. In *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*, pages 445–451. IEEE.

Osborn, J. D. and Challener, D. C. (2013). Trusted platform module evolution. *Johns Hopkins APL Technical Digest*, 32(2):536–543.

Shuja, J., Gani, A., Bilal, K., Khan, A. U. R., Madani, S. A., Khan, S. U., and Zomaya, A. Y. (2016). A survey of mobile device virtualization: taxonomy and state of the art. *ACM Computing Surveys (CSUR)*, 49(1):1.

TCG Published (2011). TPM main part 1 design principles. Specification Version 1.2 Revision 116, Trusted Computing Group.