Enclaves as accelerators: learning lessons from GPU computing for designing efficient runtimes for enclaves

Meni Orenbach, Mark Silberstein
Technion

Abstract

Intel SGX enclaves is a novel technology that holds the promise to revolutionize the way secure and trustworthy applications are built. However, from the perspective of interaction with the rest of the system, some of the enclave's characteristics are remarkably similar to the characteristics of traditional hardware accelerators, such as GPUs. For example, enclaves suffer from significant invocation overheads, offer space-constrained private memory, and cannot directly invoke OS services such as network or file I/O. Over the course of GPU computing evolution there have been developed many techniques to improve the system performance and programmability. Our key observation is that the conceptual similarities between enclaves and accelerators may help building efficient runtime support for enclaves by learning from the past experience with GPUs.

We demonstrate this simple idea by implementing SGXIO, a simple yet powerful enhancement to the current SGX runtime which boosts the performance of I/O system calls from enclaves. SGXIO design is almost identical to the design of GPUfs and GPUnet [4, 8] systems for efficient I/O services for GPU programs. Our preliminary evaluation shows that SGXIO improves the performance of a simple network parameter server for distributed machine learning by up to 3.7×. These promising results suggest new ways to design more efficient runtime and system services for enclaves.

1. Enclaves as accelerators

Intel Software Guard Extensions (SGX) introduce a mechanism for dynamically instantiating private, secure and trusted containers called *enclaves* for executing security-sensitive sections of application code [1–3, 6].

However, an isolated execution environment of the enclave is both a blessing and a curse. While the isolation is fundamental for confidentiality and trust, it implies that the enclave also blocks direct interaction of the trusted code with the rest of the system, for example, preventing access to OS I/O abstractions like files and networking services. Therefore, performing an I/O operation requires the enclave to voluntary transition into untrusted mode, perform the I/O system call and then resume its execution. As we show in the next section, these transitions between trusted and untrusted execution modes are quite expensive. The runtime support for efficient system services in enclaves is therefore the key to achieving convenient programming and high performance with SGX.

Our main observation is that although enclaves are new in commodity systems, many lessons for designing their runtime and management may be learned from the design of system support for computing accelerators like GPUs. Indeed, much like accelerators used to enhance specific functionalities (e.g., highly parallel processing in GPUs), enclaves enable efficient execution of application segments in a trusted execution environment. A closer look reveals many surprising similarities.

- Both SGX enclaves and GPUs operate in separate execution contexts from the main CPU, i.e., to initiate execution the CPU must explicitly pass instructions and data to the enclave/GPU and signal its execution. Switching in and out the secure context is costly, and may reach a few microseconds in total, as we show in the next section.
- An enclave must be invoked by the CPU via *a driver* (at least for the first initialization), much like the CPU that fully manages the GPU kernel execution. SGXv1.5 does not allow dynamic code loading and dynamic thread invocation in enclaves, just like GPUs cannot load new code and invoke it in a new GPU kernel.
- An enclave has a private memory (EPC) which forms a separate address space which is not accessible from the untrusted CPU code, similarly to the high bandwidth GPU local memory which is not directly accessible to the CPU. Here, however, GPUs went a long way to provide substantial flexibility, e.g., allow GPU memory pointers to be used in asynchronous DMA calls from the CPU.
- In order to ensure privacy, enclave's inputs and outputs must be encrypted outside the enclave, explicitly copied from/to the untrusted memory and decrypted in the enclave code. These copies are akin the staging of input/output into GPU memory for the GPU kernel execution. Here, however, the availability of GPU DMA engines and runtime support for asynchronous DMA enables higher performance than there exists in enclaves.

- An enclave may directly access untrusted CPU memory, just like the GPU may access the host memory. However this memory should not be used for saving its private execution state, much like the GPUs should avoid using the CPU memory for performance-critical memory accesses
- An enclave currently cannot dynamically change the amount of physical memory it is allocated at the initialization [1], and provides runtime support for dynamic management of that memory in the enclave. GPUs share the same memory management constraints and provide similar in-GPU memory management capabilities.
- An enclave cannot directly invoke system calls or any CPU functions, just like GPUs, and in particular, cannot manipulate its virtual address space.

We believe that these similarities are informative to understand the programming constraints which enclave programmers are facing today, and which GPU programmers have successfully overcome during the several years of GPU computing evolution. In this short paper we show, for example, that we can build efficient I/O services for enclaves by retrofitting the design of the GPU-side runtime for efficient I/O accesses from GPU kernels.

2. SGXIO: supporting low-overhead I/O from enclaves

Much like in GPU kernels, we observe a significant overhead exists whilst transitioning in and out of enclave context, named in SGX's SDK as *Ocalls*. We have measured the cost of EENTER and EEXIT to be about 7110 cycles (about 2μsec) in total on an i7 4-core Skylake CPU. For comparison, invoking a GPU kernel is up to 5μsec in NVIDIA K40 GPU, and native function call is about 200 cycles [9].

These invocation costs are incurred by every system call and every I/O operation for applications that run in an enclave, and are an impediment to running I/O intensive workloads in enclaves.

Recent work on GPU OS services [4, 8] proposed a runtime that allows I/O calls directly from GPU kernels by implementing remote procedure calls infrastructure (RPC) with the CPU backend. We use the same idea for implementing *SGXIO*, a mechanisms that enables SGX enclaves to invoke untrusted functions and system calls.

SGXIO implements an RPC infrastructure that offers *transition-less communication* between enclaves and the untrusted software. In SGXIO, all I/O calls are delegated to a separate worker thread running in parallel with the enclave in an untrusted context of the same application. An I/O request from an enclave creates a request in a queue in the untrusted memory shared with the worker thread, and waits for its completion. The worker thread detects a request and invokes the actual I/O call.

[1] Though expected to change in SGX 2.0

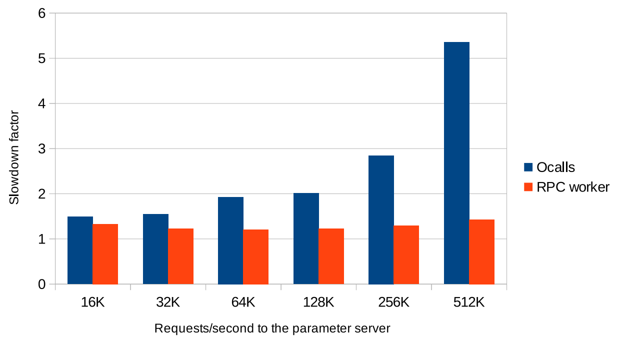

Figure 1. Requests to parameter server can scale better with RPC worker.

This architecture eliminates the transition costs associated with original Ocalls, without affecting the security guarantees of the enclaves. Specifically, the I/O request parameters and return values are passed in the untrusted memory as in Ocall.

We implemented SGXIO prototype as part of the SGX SDK, by replacing Ocalls with our RPC infrastructure. Unfortunately, as enclaves today does not support exit-less synchronization primitives, SGXIO currently implements synchronization using spinlocks via x86 atomic instructions polling a shared variable in untrusted memory to notify of new requests and computed results.

3. Preliminary results

We implement a simple parameter server commonly used in distributed machine learning systems, to maintain shared model state across a cluster of workers which run model training tasks [5]. Specifically, the server stores the model (e.g., weights of a neural network) shared across multiple workers. Each worker interacts with the server to update a certain variable or retrieve its fresh value. As both the data, and the model are considered private intellectual properties, a parameter server in an enclave is a realistic need.

Our implementation stores the model's features in a hash table which fits into the processor's *last level cache (LLC)* (4MB).

We compare the same parameter server in which network I/O is handled via Ocalls, SGXIO, or through regular function calls in clear. We run each experiment 1000 times, after 10 warmup runs, measuring only the trusted execution time, reducing noise introduced by the network and focusing on the enclave's performance. We report the average measured value, observing the standard deviation to be below 5%. We perform the evaluation on a Dell OptiPlex 7040 platform [2], running Ubuntu 14.04-64bit with the latest SGX SDK and driver.

[2] Dell OptiPlex 7040: Intel Skylake i7 4-core CPU with 8MB L3, 128 MB dedicated EPC, 16 GB RAM, and 256 GB SSD drive

Figure 1 shows the performance results of the parameter server while varying the frequency of requests to update random features. Comparing SGXIO, Ocalls and regular system calls, we observe that SGXIO is less sensitive to the request rate, thus achieve constant performance, while Ocalls suffers from the high costs of frequent transitions once the rate increases.

4. Conclusions

SGXIO is just one example of where the enclaves-as-accelerators analogy is useful. Others may include

- A DMA engine with re-encryption mechanisms to avoid enclave transitions while allowing asynchronous data transfers into enclave.
- Asynchronous enclave execution akin to GPU streams.
- Provide enclave-native virtual memory management akin to ActivePointers [7]

References

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 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, 2013.

[2] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical report.

[3] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA*, page 11, 2013.

[4] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for gpu programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, 2014.

[5] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[6] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[7] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: a case for software address translation on gpus. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 596–608. IEEE Press, 2016.

[8] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: integrating a file system with gpus. In *ACM SIGPLAN Notices*, volume 48, pages 485–498. ACM, 2013.

[9] Livio Soares and Michael Stumm. Flexsc: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 33–46. USENIX Association, 2010.