SGX-Aware Container Orchestration for Heterogeneous Clusters

Sébastien Vaucher*, Rafael Pires*, Pascal Felber*, Marcelo Pasin*, Valerio Schiavoni* and Christof Fetzer†
*University of Neuchâtel, Switzerland. E-mail: first.last@unine.ch
†Technical University of Dresden, Germany. E-mail: christof.fetzer@tu-dresden.de

***Abstract*—Containers are becoming the *de facto* standard to package and deploy applications and micro-services in the cloud. Several cloud providers (*e.g.*, Amazon, Google, Microsoft) begin to offer native support on their infrastructure by integrating container orchestration tools within their cloud offering. At the same time, the security guarantees that containers offer to applications remain questionable. Customers still need to trust their cloud provider with respect to data and code integrity. The recent introduction by Intel of Software Guard Extensions (SGX) into the mass market offers an alternative to developers, who can now execute their code in a hardware-secured environment without trusting the cloud provider.**

This paper provides insights regarding the support of SGX inside Kubernetes, an industry-standard container orchestrator. We present our contributions across the whole stack supporting execution of SGX-enabled containers. We provide details regarding the architecture of the scheduler and its monitoring framework, the underlying operating system support and the required kernel driver extensions. We evaluate our complete implementation on a private cluster using the real-world Google Borg traces. Our experiments highlight the performance trade-offs that will be encountered when deploying SGX-enabled micro-services in the cloud.

***Index Terms*—Scheduling, Intel SGX, Kubernetes, Docker.**

I. Introduction

There has been a steady trend over the last few years [1] for applications to be packaged and deployed in the form of containers, such as Docker [2], [3]. Containers offer reproducible execution environments, light lifecycle management, and closer-to-metal performance than classical virtual machine deployments. Container registries—public ones such as Docker Hub [4], or private ones such as Google Cloud Registry [5]—hold and serve thousands of container images, ready to be used. Hence, developers can simply rely on cloud infrastructures to deploy services of various nature. As such, containers are used for all kind of services, from simple key-value stores (*e.g.*, Redis [6]), to web servers (Apache [6]); from full-fledged relational databases (MySQL [7]) to scalable discovery services [8]. Major cloud operators natively support container deployments in their infrastructure (*e.g.*, Google Container Engine [9] or Amazon EC2 Container Service [10]). Similarly, it is straightforward to set up a container cluster on private premises, leveraging popular [11] container orchestrators (or schedulers) such as Kubernetes or Docker Swarm.

Without special care, containers are exposed to critical security threats. For instance, the cloud infrastructure could be compromised by malicious actors or software bugs. This is especially true when containers are deployed over public cloud infrastructures, but it also holds true for the case of a deployment on private premises that may have been corrupted by malicious actors or software. Hence, service deployers are left with no other choice than to trust the infrastructure provider and the complete software stack (including the underlying operating system, kernel libraries, etc.). Similarly, they must face the risk that a compromised component can lead to severe data leakage [12], [13].

To overcome these issues, one could adopt solutions based on homomorphic cryptosystems [14]. However, their performance is several orders of magnitude slower than native systems, and as of today, these solutions are still impractical for real-world deployment and adoption [15].

The introduction of Intel Software Guard Extensions (SGX) [16] into the mass market with the Intel Skylake family of processors [17] offers a promising alternative. The availability of SGX allows the deployers to distrust the cloud operator and rely instead on hardware protection mechanisms, hence drastically reducing the Trusted Computing Base (TCB). Moreover, programs execute at almost native speed [18]. Cloud providers are starting to offer SGX-enabled Infrastructure as a Service (IaaS) solutions to end-users. One example is Microsoft Azure Confidential [19]. We expect other cloud providers to introduce similar offerings in the short-term.

Deploying and orchestrating containers on a heterogeneous cluster, with a mix of machines with and without SGX capabilities, presents its own set of specific challenges. The containers that require SGX will contend on the availability of dedicated memory (see Section II). Therefore, the monitoring infrastructure that feeds the scheduler master with resource usage metrics must keep track of the demanded SGX memory requests and schedule the containers accordingly. Unfortunately, none of the existing container orchestrators offer native support to provide runtime insights about the resources used by SGX containers. Notably, all of them rely on statically-provided information given by the users upon deployment. This information can be malformed or non-conforming to the real usage of the containers, and henceforth leading to over- or under-allocations.

In the proposed context, our contributions are the following. We propose an SGX-aware architecture for orchestrating containers. Fitting in this architecture, we offer an open-source vertical implementation [20] of the required system support, including modifications to the Linux driver for SGX as well

as a Kubernetes *device plugin*. Further, we show that it will be possible to drastically reduce the waiting time of the submitted jobs by exploiting future versions of SGX, as they will offer better control over the size of the dedicated memory. Finally, we demonstrate that our design and implementation are sound with a detailed evaluation using the Google Borg traces [21].

The rest of the paper is organized as follows. We provide a short introduction to Intel SGX in Section II. In Section III, we describe our trust model and assumptions. We describe the architecture in Section IV and our prototype implementation in Section V, followed by its evaluation in Section VI. We survey the related work in Section VII, before concluding and describing how we plan to extend this work in Section VIII.

II. Background on Intel SGX

The design of our system revolves around the availability of Intel SGX on hosts. It consists in a Trusted Execution Environment (TEE) recently introduced by the Skylake family of Intel processors, similar in spirit to ARM TrustZone [22]. Applications create secure *enclaves* to protect the integrity and confidentiality of the code being executed with associated data.

The SGX mechanisms are depicted in Figure 1. They allow applications to access and process confidential data from inside the enclave. The architecture guarantees that an attacker with physical access to a machine cannot tamper with the application data or code without being noticed. Essentially, the Central Processing Unit (CPU) package represents the security boundary. Moreover, data belonging to an enclave is automatically encrypted, for confidentiality, and its digests inserted in a hash tree, for integrity and freshness. A memory dump on a victim's machine will only produce encrypted data. A custom *remote attestation protocol* allows to verify that a particular version of a specific enclave runs on a remote machine, using a genuine Intel processor with SGX enabled. An application using enclaves must ship a signed (not encrypted) shared library (*i.e.*, a shared object file in Linux) that can possibly be inspected by malicious attackers. Data stored in enclaves can be saved to persistent storage, protected by a seal key. This allows to store sensitive data on disk, waiving the need for a new remote attestation every time the SGX application restarts.

There are three memory areas in which data relative to an SGX enclave may be stored [16]. Data inside the CPU package (registers, level 1-3 caches, etc.) is stored in plaintext, as long as the processor is in enclave mode and executes the enclave owning the data. SGX enclaves can also use a dedicated subset of system memory called the Enclave Page Cache (EPC). The EPC is split into pages of 4 KiB and exists in Processor Reserved Memory (PRM), a range of system memory that is inaccessible to other programs running on the same machine, including privileged software such as the operating system. It is a small memory area; current hardware supports at most 128 MiB. This size is configurable via Unified Extensible Firmware Interface (UEFI) parameters, but a reboot is required to apply the change. Note also that the EPC is shared among all the applications executing inside enclaves,

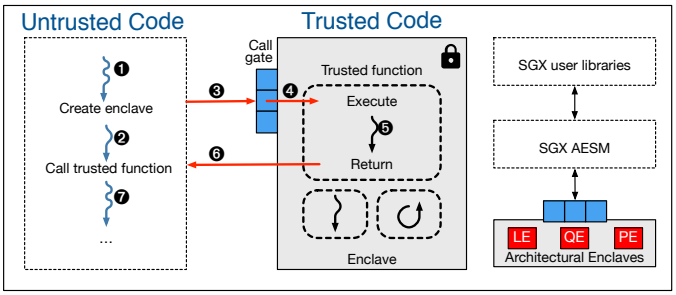

Fig. 1. SGX core operating principles.

hence being a very scarce and highly contended resource. Future releases of SGX will relax this size limitation [23]. Our evaluation shows, by means of simulation, how changes in the next version of SGX would impact the performances of a scheduler, for example in terms of turnaround time for submitted jobs (see Subsection VI-D).

Only 93.5 MiB out of 128 MiB can effectively be used by applications (for a total of 23 936 pages), while the rest is used for storing SGX metadata. In order to provide more flexibility, SGX implements a paging mechanism. It allows to page-out portions of trusted memory into regular system memory. Any access to an enclave page that does not reside in the EPC triggers a page fault. The SGX driver interacts with the CPU to choose the pages to evict. The traffic between the CPU and the system memory is protected by the Memory Encryption Engine (MEE) [24], also in charge of providing tamper resistance and replay protection. If a cache miss hits a protected region, the MEE encrypts or decrypts data before sending to, respectively fetching from, the system memory and performs integrity checks.

The execution flow of a program using SGX enclaves works as follows. First, an enclave is created (see Figure 1-❶) by the *untrusted* part of the application. It must then be initialized using a *launch token*. Intel's Launch Enclave (LE) can help in fetching such token. Access to the LE and other architectural enclaves, such as the Quoting Enclave (QE) and the Provisioning Enclave (PE) is provided by the Intel Application Enclave Service Manager (AESM). SGX libraries provide an abstraction layer for communicating with the AESM. As soon as a program needs to execute a trusted function (see Figure 1-❷), it executes the `ecall` SGX primitive (❸). The call goes through the SGX call gate to bring the executing thread inside the enclave (❹). After the function is executed in the trusted environment (❺), it calls the return instruction (❻), before giving the control back to the caller (❼).

III. Trust Model

We assume that our SGX-enabled orchestrator is deployed on the premises of a given cloud provider. Providers show an honest-but-curious behavior. They are interested in offering an efficient service to customers, mostly for selfish economic reasons (*e.g.*, providers want to maximize the number of executed jobs per unit of time, but they will not deliberately disrupt them). However, providers do not trust their customers,

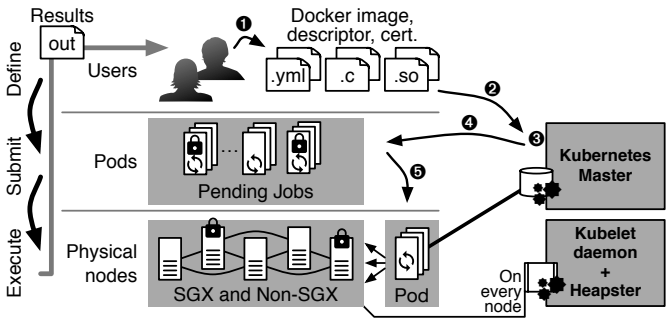

Fig. 2. Architecture and workflow of the SGX-aware scheduler.

especially not the resource usage declarations they specify at deployment time. Nevertheless, providers trust their own infrastructure, namely the operating system running on the cluster nodes, the SGX driver, as well as Kubernetes itself.

Customers rely on the infrastructure offered by cloud providers, but do not trust it. Notably, they trust Intel SGX to prevent providers from inspecting the contents or tampering with jobs deployed on their infrastructure. In turn, customers might try to allocate more resources than what they requested, either at the cost of providers or their tenants.

In previously existing cloud deployments, customers had to blindly trust providers not to pry into their jobs, nor tamper with them. Conversely, providers had to trust that their customers do not overstep their share of EPC, as they had no means to limit it. Thanks to the novel work described in this paper, providers can now effectively supervise SGX-related resources by enforcing EPC allocation limits, thence rendering SGX cloud deployments practical. As a consequence, customers can enjoy truthful job execution in the cloud without having to trust their provider.

IV. Architecture

This section describes the architecture of our system, as depicted in Figure 2. The complete system implements a container orchestrator that can efficiently schedule SGX-enabled jobs, as well as regular jobs, on a heterogeneous cluster. We assume that SGX-enabled jobs execute entirely in enclaves, minus a part responsible for bootstrapping the SGX enclaves. These assumptions match those of state-of-the-art SGX systems, *e.g.*, SCONE [18] or Graphene-SGX [25].

One important aspect of the system is its ability to guarantee that jobs submitted to a given host always fit within its EPC memory limits. This is of particular relevance to avoid major performance penalty [18]. To achieve this goal, we use a monitoring layer, implemented by means of daemons executing on each node of the cluster. They accurately measure individual EPC utilization of the jobs submitted to our system. These measures are periodically sent to a central node, in charge of storing and analyzing them. We rely on a time-series database for this task, as they have proved to be an efficient tool to implement queries over moving sliding windows [26]–[28], such as the ones we use in our scheduling policy.

Users submit their jobs by specifying the name of the container image (see Figure 2-❶). They need to indicate the amount of EPC memory required by their jobs at this time.[1] Apart from these parameters, clients rely on the regular Application Programming Interfaces (APIs) of the orchestrator. The image is initially pulled from a public or private container registry. Afterwards, it is submitted into the scheduler's queue as a pending job (❷).

Scheduling algorithm. Our scheduling algorithm works as follows. First, the container orchestrator fetches the list of pending jobs currently in the queue (❸). It takes their memory allocation requests into account, both in terms of standard memory and EPC. At the same time, it fetches accurate, up-to-date metrics about memory usage across all nodes in the cluster. This is done by executing a sliding-window query over the time-series database. The scheduler then combines the two kinds of data to filter out job-node combinations that cannot be satisfied, either due to hardware compatibility (*i.e.*, SGX-enabled job on a non-SGX node), or if the job requests would saturate a node. The next step depends on the concrete container placement policy configured for the cluster. In particular, we enable support for SGX measures in two well-known placement policies.

When *binpack* is in use, the scheduler always tries to fit as many jobs as possible on the same node. As soon as its resources become insufficient, the scheduler advances to the next node in the pool. The order of the nodes stays consistent by always sorting them in the same way. In the case of a standard job, we sort SGX-enabled nodes at the end of this list, to preserve their resources for SGX-enabled jobs.

Conversely, the main goal of the *spread* strategy is to even out the load across all nodes. It works by choosing job-node combinations that yield the smallest standard deviation of load across the nodes. Like *binpack*, it only resorts to SGX-enabled nodes for non-SGX jobs when no other choice is possible to execute the job. After the policy selection is made, the scheduler communicates the computed job-node assignments to the orchestrator (❹). It then handles the actual deployment of jobs towards the various nodes (❺).

There may be jobs that cannot be fitted in the cluster at the time of their submission. The orchestrator keeps a persistent queue of pending jobs (❸); the scheduler periodically checks for the possibility to schedule some of them, applying a first-come first-served (FCFS) priority.

V. Implementation Details

In this section we provide insights on the implementation of the components of our architecture. Although several mainstream container orchestrators exist today, we decided to build our implementation on top of the open-source Kubernetes[2]

[1] As the allocation of EPC memory has to be done at program initialization, this value is hard-coded in the SGX-enabled binary and could be extracted directly from it. We rely on the user's specification for convenience reasons.

[2] We also report that our initial attempt was based on Docker Swarm. However, its lack of hooks to extend the architecture and poor developer documentation convinced us to look for a different solution.

container orchestrator [29]. Likewise, the entire source code of our implementation is released as open-source software [20]. The components that we add to Kubernetes' architecture interact with it using its public API. This approach further facilitates the integration into future versions of Kubernetes.

In the remainder of this section, we provide implementation details for: our *device plugin* (Subsection V-A), our custom SGX-aware scheduler (Subsection V-B), and how we implemented SGX metric probes (Subsection V-C). We then highlight how we enforce SGX-related resource usage limits (Subsection V-D). Moreover, we describe our extensions to the Intel SGX driver for Linux (Subsection V-E). Finally, we present our base Docker image that SGX application developers can use as a base for their applications (Subsection V-F).

A. Kubernetes device plugin

The first component is a *device plugin* that allows to mark a Kubernetes node as able to execute SGX instructions. The plugin can be used directly by the Kubelet [29] node agent, since Kubernetes 1.8 [30]. Its original design was motivated by the developers' need to access Graphics Processing Units (GPUs). Its intent is to expose system devices (*i.e.*, those available in the `/dev` pseudo-filesystem) directly within Kubernetes. Opportunely, applications implemented using the Intel Software Development Kit (SDK) for Linux are given access to SGX by means of a pseudo-file in `/dev`. Communication between Kubelet and the *device plugin* leverages Google Remote Procedure Calls (gRPC) [31]. Our *device plugin* checks for the availability of the Intel SGX kernel module on each node and reports it to Kubelet. Kubelet notifies the master node about the availability of an "SGX" resource on that node.

The philosophy behind *device plugins* is to register one resource item (*e.g.*, one graphics processing card, one FPGA board, etc.) per physical device. In the case of SGX applications, there is only one pseudo-file registered per processor. However, SGX allows multiple enclaves to be executed at the same time, sharing the EPC. Exposing only one resource item would have utterly limited the usefulness of our contribution, as only one SGX-enabled set of containers (or *pod* in the Kubernetes terminology) could have been scheduled on a physical host at any given time. We solve this problem by exposing each EPC page as a separate resource item. By exposing the EPC as multiple independent "devices", several pods can be deployed and share a single node, thus supporting the execution of several SGX applications at once. Despite the great amount of resources created with this scheme, we did not notice any perceptible negative influence on performance.

Although SGX allows over-commitment of its protected memory via paging, doing so leads to severe performance drops up to $1000\times$ [18]. Therefore, we deliberately prevent over-commitment of the EPC, in order to preserve predictable performance for all pods deployed in the cluster.

Kubernetes will then proceed and mount the SGX device file inside each pod that requested at least one share of EPC. Therefore, end-users must declare that their SGX-enabled pods use some amount of the "SGX" resource. In the case of Kubernetes, this is done by filling in the resource requests and limits fields of the pod specification. Resource requests are used by the scheduler to dispatch SGX-enabled pods towards a suitable node. Limits are transmitted to our modified SGX driver for strict enforcement (see Subsection V-D).

```
SELECT SUM(epc) AS epc FROM
  (SELECT MAX(value) AS epc FROM "sgx/epc"
    WHERE value <> 0 AND time >= now() - 25s
    GROUP BY pod_name, nodename
  )
GROUP BY nodename
```

Listing 1. InfluxQL query.

B. SGX-Aware Scheduler

Once nodes have been configured with our *device plugin*, they are ready to accept pods using Intel SGX instructions. However, Kubernetes' own scheduler only relies on values communicated in the resource requirements of each pod. Given the restrained capacity of the EPC, it is imperative to maximize its utilization factor. To do so, the scheduler must consider the actual usage metrics of the cluster, collected at runtime. Our scheduler can decide on scheduling actions based on actual measured memory usage (for the EPC as well as regular memory). Metrics regarding the regular memory are collected by Heapster [32], while SGX-related metrics are gathered using our custom probes (see Subsection V-C).

We implemented the previously described scheduling strategies (*binpack* and *spread*) in a non-preemptive manner, following the same scheme as Kubernetes' default scheduler. The scheduler itself is packaged as a Kubernetes pod. This allows it to execute with the same privileges as the default scheduler. It also provides us with seamless migrations and crash monitoring features, as for any pod.

Kubernetes supports multiple schedulers to concurrently operate over the same cluster. It is therefore possible to deploy our scheduler with both of its strategies, in parallel to the default, non SGX-aware one. Comparative benchmarking is thus made easier, as each pod deployed to the cluster can specify which scheduler it requires. We assume that, in production deployments, only one variant of our SGX-aware scheduler will be deployed as default scheduler to prevent conflicts between schedulers.

C. SGX metrics probe

The proposed scheduling algorithm relies on metrics directly fetched from nodes of the cluster. Kubernetes natively supports Heapster [32], a lightweight monitoring framework for containers. We configured Heapster to collect such metrics on each node and subsequently store them into an InfluxDB [33] time-series database. Then, the system performs InfluxQL queries [34] against the database. Listing 1 shows how to report the total size of EPC memory used in the last 25 seconds per pod, then grouped by node and summed up.

We have implemented an SGX-aware *metrics probe* to gather EPC usage metrics from our modified Intel SGX driver

(see Subsection V-E). These metrics are pushed into the same InfluxDB database used by Heapster. This allows our scheduler to use equivalent queries for SGX- and non SGX-related metrics.

The probe is deployed on all SGX-enabled nodes using the *DaemonSet* component [35]. The distinction between standard and SGX-enabled cluster nodes is made by checking for the EPC size advertised to Kubernetes by the *device plugin*. Finally, we leverage Kubernetes itself to automatically handle the deployment of new probes when adding physical nodes to the cluster, as well as their management in case of crashes.

D. Enforcing limits on EPC usage

SGX allows over-commitment of its primary cache, the EPC. However, this feature comes with an important performance penalty for user applications. It is imperative for a cloud provider that wants to guarantee a fair share of resources to make sure that multiple containers share the EPC in a respectful way. In Kubernetes, the users specify the limits for each type of resource that their pods use. These values are later used for several purposes, *e.g.*, accounting and billing the reserved resources. It is in the interest of the infrastructure provider to make sure that co-hosted containers do not contend on the same resource. A user with malicious intents could advertise lower amounts of resources than what his pods actually use. For this reason, it is crucial to enforce the limits advertised in the specification of each pod. In our particular context, we focus on limits related to the EPC usage.

We implement proper limits enforcement by modifying two existing components of our architecture: (i) the SGX driver provided by Intel (see Subsection V-E), and (ii) Kubelet, the daemon running on each node of a Kubernetes cluster. The SGX driver will deny the initialization of any enclave that exceeds the share of pages advertised by its enclosing pod.

Linux cgroups. The proper way to implement resource limits in Linux is by adding a new *cgroup* controller to the kernel [36]. This represents a substantial engineering and implementation effort, affecting several layers of our architecture. Modifications would be required in Kubernetes, Docker (which Kubernetes uses as container runtime) and the Linux kernel itself.

We considered a simpler, more straightforward alternative. Namely, we use the *cgroup* path as a pod identifier. The rationale behind the choice of this identifier is as follows: (i) it is readily available in Kubelet and in the kernel; (ii) all containers in a pod share the same *cgroup* path, but distinct pods use different ones; (iii) the path is available before containers actually start, so this allows the driver to know the limits applicable to a particular enclave on its initialization.

In order to communicate limits from Kubernetes to the SGX driver, we added 16 lines of Go code and 22 lines of C code to Kubelet, using *cgo* [37]. These additions communicate a new *cgroup path–EPC pages limit* pair each time a pod is created. Finally, the communication channel uses a new input/output queryable (*ioctl*) [38] added to the SGX driver detailed in Subsection V-E.

E. Modified Intel SGX Driver

We modified the Linux kernel driver [39] provided by Intel. Our modifications revolve around two closely-related topics: gathering usage data to improve scheduling decisions, and enforcement of resource usage limits. We offer access to the total number of EPC pages, as well as their current usage status by way of module parameters. They are accessible using the usual Linux filesystem interface, below the `/sys/module/isgx/parameters` path. Values can be retrieved through two pseudo-files: `sgx_nr_total_epc_pages` (total amount of pages on the system) and `sgx_nr_free_pages` (amount of pages not allocated to a particular enclave).

Additionally, the EPC usage can be probed at a finer granularity, in a per-process manner. To do so, we created a new input/output queryable using the `ioctl` function [38] available in Linux. This control reports the number of occupied EPC pages given to a single process, described by its *process identifier*. This metric is helpful to identify processes that should be preempted and possibly migrated, a feature especially useful in scenarios of high contention.

We created a second *ioctl* to communicate resource usage limits (see Subsection V-D). Each pod deployed in our cluster needs to advertise the number of EPC pages it plans to use. This number is then shared between Kubelet and the driver by issuing an *ioctl* at the time of pod creation. The driver makes sure that limits can only be set once for each pod, therefore preventing the containers themselves from resetting them. In the current version of SGX, enclaves must allocate all chunks of protected memory that they plan to use at initialization time. We add a couple lines of code in `__sgx_encl_init` to call a function that checks whether to allow or deny a given enclave initialization. Internally, it compares the number of pages owned by the enclave to the limits advertised by its enclosing pod.

The implementation of these features consists in 115 lines of C code on top of the latest Intel SGX release for Linux [40].

F. Base Docker Image and Intel SDK for SGX

When an application wants to use the processor features offered by Intel SGX, it has to operate in enclave mode. Before it can switch to this mode, the program has to pre-allocate all the enclave memory that it plans to use. The particular x86 instructions that can reserve enclave memory can only be executed by privileged code running in *ring 0* [16]. Under the GNU/Linux operating system, only the kernel and its modules are allowed to call these instructions. In a containerized context, the kernel and, by extension, its modules are shared across all containers. In the case of running SGX-enabled applications in containers, this implies that it is required to set-up a communication channel between the container and the *isgx* module. In Docker, this can be done by mounting the `/dev/isgx` pseudo-file exposed by the host kernel directly into the container. While it is possible to create SGX-enabled applications that directly interface with the kernel module, the Intel SGX SDK provides an easier path

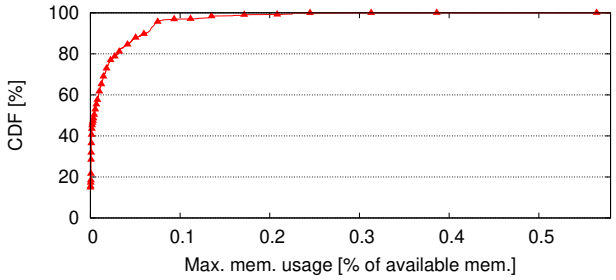

Fig. 3. Google Borg trace: distribution of maximal memory usage.

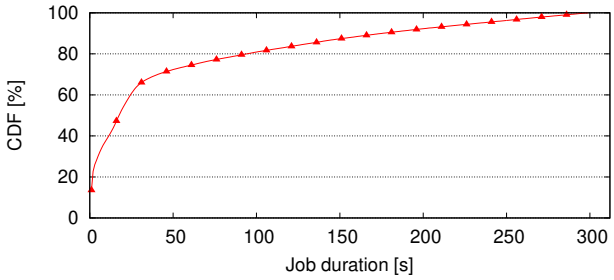

Fig. 4. Google Borg trace: distribution of job duration.

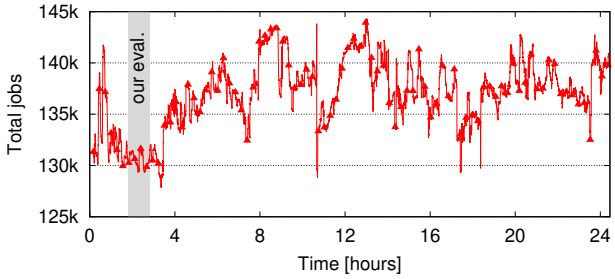

Fig. 5. Google Borg trace: concurrently running jobs during the first 24 h.

to SGX application development. Programs that are using this SDK rely on the Platform Software (PSW) [41]. Hence, we created a Docker image that allows SGX-enabled applications developed using the official SDK to be executed seamlessly within Docker containers. The image is publicly available from Docker Hub [42].

VI. Evaluation

This section reports on a detailed evaluation of our SGX-aware scheduler and its subcomponents. First, we describe our experimental settings in Subsection VI-A. Then, in Subsection VI-B, we characterize the Google Borg trace and the simplifications done to adapt it to the scale of our cluster. We characterize the synthetic workload that we use in Subsection VI-C. Subsection VI-D shows the performance penalties induced by SGX. A comprehensive evaluation of the scheduler itself follows in Subsection VI-E, with Subsection VI-F concentrating on measuring the effectiveness of strictly enforcing resource usage limits. We end our evaluation by investigating how our work will be affected by the release of SGX 2 (Subsection VI-G).

A. Evaluation settings

Our cluster consists of 5 machines. The first 3 machines are Dell PowerEdge R330 servers, each equipped with an Intel Xeon E3-1270 v6 CPU and 64 GiB of RAM. One of these machines acts as Kubernetes master, while the remaining are regular Kubernetes nodes. The two remaining nodes are SGX-enabled machines, also acting as nodes in the Kubernetes cluster. These machines feature an Intel i7-6700 CPU and 8 GiB of RAM. SGX is statically configured to reserve 128 MiB of RAM for the EPC. The machines are connected to a 1 Gbit/s switched network. We use Kubernetes (v1.8), installed on top of Ubuntu 16.04. We enabled the Kubernetes *device plugin* alpha feature on all the machines.

B. The Google Borg Trace

Our evaluation uses the Google Borg Trace [21], [43]. The trace was recorded in 2011 on a Google cluster of about 12 500 machines. The nature of the jobs in the trace is undisclosed. We are not aware of any publicly available trace that would contain SGX-enabled jobs. Therefore, we arbitrarily designate a subset of trace jobs as SGX-enabled. In the following experiments, we insert various percentages of SGX jobs in the system: from 0 % of SGX jobs (only standard jobs), and then increasing by 25 % steps, until 100 % (only SGX jobs).

The trace reports several metrics measured for the Google jobs. We extract the following metrics out of it: *submission time*, *duration*, *assigned memory* and *maximal memory usage*. The submission time is crucial to model the same arrival pattern of the jobs in our cluster. The run time of each job matches exactly the one reported in the trace, set aside system clock skews. We use the assigned memory as the value advertised to Kubernetes when submitting the job to the system. However, the job will allocate the amount given in the maximal memory usage field. We believe this creates real-world-like behavior *w.r.t.* the memory consumption advertised on creation compared to the memory that is actually used.

The trace specifies the memory usage of each job as a percentage of the largest memory capacity in Google's cluster (without actually reporting the absolute values). In our experiments, we set the memory usage of SGX-enabled jobs by multiplying the memory usage factor obtained from the trace to the total usable size of the EPC (93.5 MiB in our case). As for standard jobs, we compute their memory usage by multiplying them to 32 GiB. The rationale behind this choice is that it is the power-of-2 closest to the average of the total memory installed in our test machines. Moreover, we think that it yields amounts that match real-world values. Figure 3 shows the amounts of memory allocations recorded in the trace. Given the size of Google's cluster, we have to scale down the trace before being able to replay it on our own cluster setup. We scale the trace down along two dimensions.

Time reductions. Figure 4 shows the Cumulative Distribution Function (CDF) of the duration of jobs found in the trace. All jobs last at most 300 s. Hence, 1 h is sufficiently long to properly stabilize the system. Instead of considering the full

29 days recorded in the trace, we use a 1-hour subset ranging from 6480 s to 10 080 s extracted from the first 24 hours of execution (highlighted in light gray in Figure 5). This slice of trace, while being the less job-intensive in terms of concurrent jobs for the considered time interval, still injects an intensive load on the cluster. Hence, we also operate a frequency down-scaling of it, described next.

Frequency reductions. We sample every 1200th job from the trace, to end up with a number of jobs big enough to cause contention in the system, but that does not clutter it with an incommensurable amount of jobs. Figure 5 displays the concurrent amount of running jobs recorded in the trace, before sampling.

C. Matching trace jobs to deployable jobs

After processing the trace file, we get a timed sequence of jobs with their effective memory usage. In order to materialize this information into actual memory or EPC usage, our jobs are built around containers that run STRESS-SGX [44], a fork of the popular STRESS-NG [45] stress tool. Normal jobs use the original virtual memory stressor brought from STRESS-NG, while SGX-enabled jobs use the topical EPC stressor. We specify parameters to allocate the right amount of memory for every job, in accordance with the values reported by the trace.

D. Evaluation of SGX performance

The main sources of overhead for SGX enclaved executions are the transitions between protected and unprotected modes, and memory usage [18], [46]–[48]. Additionally, startup time is longer than traditional executions, mainly due to support service initialization and memory allocation. The Intel SDK [41] provides the Platform Software (PSW) that includes the Application Enclave Service Manager (AESM). As the name suggests, it is a service that eases the process of deploying enclaves, performing common tasks such as attestation and supporting the access to platform services, like obtaining trusted time and monotonic counters.

By default, Docker—and more in general containers based on *cgroups*—enforces strict limitations to what programs running inside the container can get access to. It especially isolates the host from potentially malicious containers. The isolation layer can be lifted by running containers in *privileged* mode, a potentially dangerous and risky operation [49], especially in a shared computing environment. As we want to keep containers isolated, we need to have one instance of the PSW running in each container. Therefore, each SGX-enabled containerized process will suffer from a small initialization penalty, as it will need to wait for the PSW to initialize before it can start its useful work.

An additional startup overhead is due to the enclave memory allocation, since all of it must be committed at enclave build time, to be measured for attestation purposes [23].

We start by quantifying the overhead for launching AESM and allocating memory. Figure 6 shows the required average time required for 60 runs. Error bars represent the 95 % confidence interval. We omit measurements for standard jobs

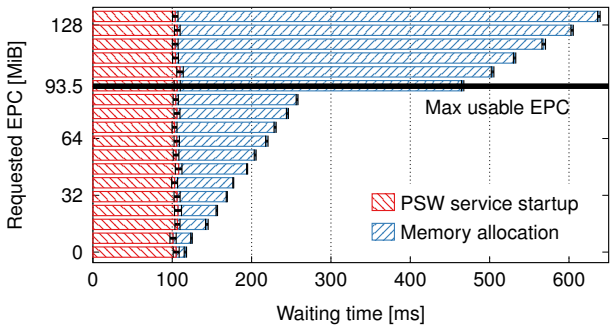

Fig. 6. Startup time of SGX processes observed for varying EPC sizes.

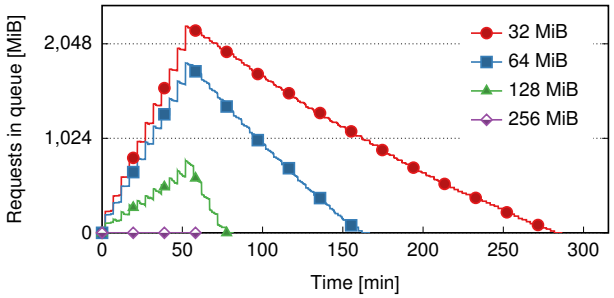

Fig. 7. Time series of the total memory amount requested by pods in pending state for different simulated EPC sizes.

since they steadily took less than 1 ms, negligible compared to SGX ones. As expected, the service startup time is virtually the same in all runs, accounting for about 100 ms. On the other hand, memory allocation time shows two clear linear trends: before and after reaching the usable EPC memory limit. Until this limit, the time increase rate is 1.6 ms/MiB after which it jumps to 4.5 ms/MiB, plus a fixed delay of about 200 ms. Note that these times are just for *allocating* memory, still without any real use. Even higher overheads are expected when processes use it [18]. This experiment reinforces the importance of accurate SGX job scheduling to circumvent the soft-limits imposed by the EPC size.

Next, we evaluated the trace execution with different EPC sizes. This particular run is based on simulation, but uses the exact same algorithms and behaves in the same way as our concrete scheduler. It allows us to operate with various EPC sizes, including those that will be available with future SGX hardware. Figure 7 shows the amount of memory requested by pods in pending state along the time, with varying maximum reserved memory. On the x-axis, we have the duration of the trace, while on the y-axis we see the total amount of EPC size requested by queued jobs that could not be immediately scheduled. Looking at the extremes, we can notice the total absence of contention when the EPC accounts for 256 MiB, finishing the batch execution in one hour, exactly as recorded in the trace. Conversely, the trace takes 4 h 47 min when the maximum EPC memory usage is 32 MiB. In between, a 64 MiB EPC would allow the trace execution to finish after 2 h 47 min. For 128 MiB, the maximum EPC limit of current processors, the batch would conclude after 1 h 22 min.

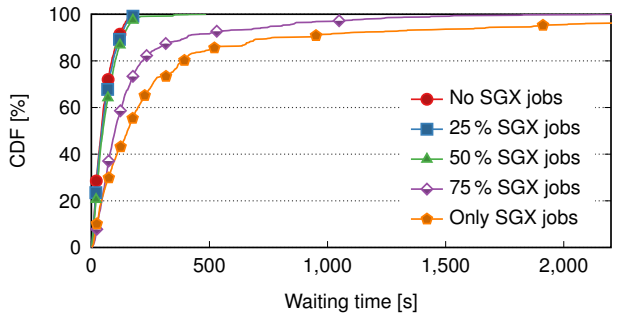

Fig. 8. CDF of waiting times, using varying amounts of SGX-enabled jobs.

This experiment, despite being dependent on this particular job trace, puts in evidence the benefits of having bigger protected memory sizes. Having a processor capable of using 256 MiB of EPC is theoretical at this stage. Nonetheless, we expect SGX 2 (see Subsection VI-G) to allow larger EPC sizes, and consequently make SGX more appealing to cloud providers.

E. Scheduler evaluation

We conclude the evaluation of our SGX-aware scheduler by replaying the trace described in Subsection VI-B. All performance metrics are directly fetched from Kubernetes.

Figure 8 shows the CDF of waiting times observed by jobs before their execution. The waiting time refers to the period between the submission of the job to the orchestrator, and the instant when the job actually starts on a given node of the cluster. In this experiment, we use the *binpack* scheduling strategy. As expected, the run that only uses standard memory (no SGX jobs) experiences relatively low waiting times. On the other side, the pure SGX run waiting times go off the chart, due to much higher contention conditions. The longest wait observed by a job is 4696 s, more than the total task duration given in the trace. When 25 % to 50 % of the jobs are SGX-enabled, waiting times are really close to the ones observed with a fully native job distribution. This shows that incorporating a reasonable number of SGX jobs has close to zero impact on the scheduling. Notably, we expect real-world deployments to include small percentages of jobs requiring SGX instructions, although this might change in the future.

Figure 9 depicts the waiting times observed in relation to the amount of memory requested by pods. The top plot shows the results for the *spread* strategy, while *binpack* is shown at the bottom. There are two rows of labels in the x-axis. The top row with smaller values is applicable to SGX jobs while the bottom row is applicable to standard jobs. All values are extracted from the same run with a 50 % split between standard and SGX jobs. The error bars are computed using the 95 % confidence interval. We can observe that the *spread* strategy is consistently worse than *binpack*. Binpack also seems to handle bigger memory requests better. SGX jobs show similar waiting times compared to standard jobs, save for one outlier in the *binpack* plot. This shows that our scheduler works well with both types of jobs.

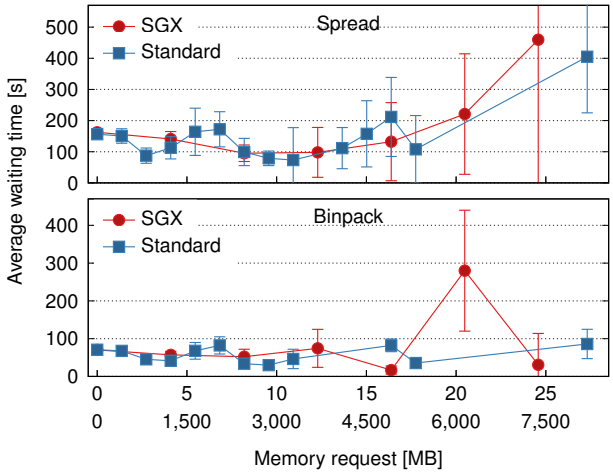

Fig. 9. Waiting times for SGX and non-SGX jobs, using *binpack* and *spread* scheduling strategies, depending on the memory requested by pods.

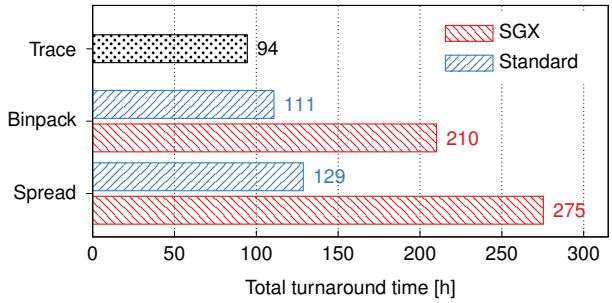

Fig. 10. Sum of turnaround times for all jobs sent to the cluster, compared with the time reported by the trace.

Finally, Figure 10 shows the aggregated turnaround time for all jobs. This metric refers to the duration elapsed between the instant a job was submitted to the moment when the job finishes and dies. At the top of the figure, the dotted black bar (labeled "Trace") represents the total useful job duration, as recorded in the trace. The difference with the other results highlights the total waiting time for each of the different settings. We use runs that only contain one type of job (either all SGX or regular jobs). As we noted in Figure 9, the *binpack* strategy, in this specific setting and portion of the trace, achieves the best result (shorter turnaround time). When using the *binpack* strategy, SGX jobs need slightly less than twice the time of their non-SGX counterparts. The total waiting time difference between the two kind of jobs is above this ratio, but the impact on the total turnaround time is limited to some extent. Although a more in-depth evaluation of the trade-offs between the *binpack* and *spread* scheduling strategies would allow for a more comprehensive understanding of our setting, we believe that individual workload characteristics are the key factors when selecting a placement strategy.

Our decision to choose a multiplier of 32 GiB for standard jobs and 93.5 MiB for SGX-enabled jobs (see Subsection VI-B) considerably affects the performance difference between standard and SGX jobs. Indeed, our whole cluster

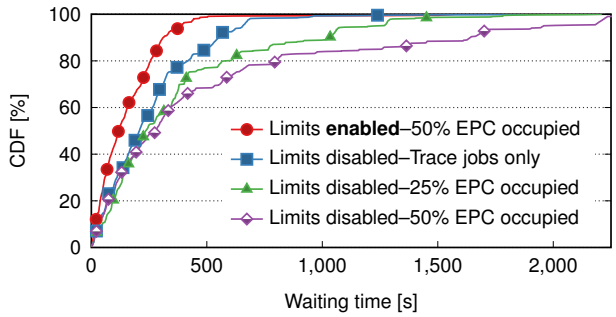

Fig. 11. Observed waiting times when malicious containers are deployed in the system, with and without usage limits being enforced.

has $2 \times 93.5\,\mathrm{MiB} = 187\,\mathrm{MiB}$ of EPC memory (on the two SGX-enabled machines) compared to a total amount of $2 \times 64\,\mathrm{GiB} + 2 \times 8\,\mathrm{GiB} = 144\,\mathrm{GiB}$ of regular system memory. This represents a difference of almost 3 orders of magnitude ($788\times$) between the two kinds of memory, whereas the difference between the scaling multipliers is only half of that ($350\times$). Therefore, in relative terms, SGX jobs have close to $2\times$ less memory at their disposal compared to standard jobs. As highlighted in Subsection VI-D, doubling the amount of memory drastically improves performance, which explains the performance gap. We observe nonetheless that using a $50\,\%$ split between standard and SGX jobs yields acceptable performance for both kinds of jobs.

F. Impact of resource usage limits

One of the feature of our system is the strict enforcement of limits regarding per-container EPC memory consumption (see Subsection V-D). Resource usage limits are declared by the users themselves, and therefore could be inaccurate with regard to the real usage made by their containers. We identify incentives to lead users into truthfully declaring resource usages: if the user declares too high a limit for his container, then the infrastructure provider will charge him for the additional resources. On the other hand, declaring too low resource usages will lead to the container being denied service due to the enforcement of limits. To show the potential damages that users could cause without this mechanism, we ran the same experiment as in Subsection VI-B, but we add so-called malicious containers to the system. We deploy as many of them as there are SGX-enabled nodes in the cluster. The *modus operandi* of these containers is to declare 1 page of EPC as limit and request in their pod specification, but actually use way more: up to $50\,\%$ of the total EPC available on the machine they execute on.

The results, as presented in Figure 11, show that, without strict limits being enforced, most honest containers in the system suffer from longer waiting times. Obviously, as the size of the allocations made by malicious containers increases, the effects suffered by honest containers grow as well. Fortunately, we clearly see that enforcing limits on memory allocations annihilates the efforts of the malicious containers. The reason why the run with malicious containers and limits enabled is better than the one that is just following the trace is because some jobs in the Google Borg trace actually try to allocate more memory than they advertise (44 jobs out of 663 show this behavior). When we strictly enforce memory limits, these jobs are immediately killed after launch.

G. Compatibility with SGX 2

The hardware in current Intel processors only supports version 1 of SGX. Intel has already published several design documents regarding the second version, SGX 2 [23], [50]. The SDK and the driver adapted to SGX 2 have been recently published [41]. The most important feature that this new version introduces is dynamic EPC memory allocation. Enclaves can ask the operating system for the allocation of new memory pages, and may also release pages they own. Contrary to the current version, these operations can also be done during their execution. Considering the limited amount of EPC that is shared by every enclave running on the same node, this new feature can really improve resource utilization on shared infrastructures.

As far as our scheduler is concerned, we believe that only minor changes need to be performed to fully take these new possibilities into account. Provided that Kubernetes nodes are deployed on SGX 2-compatible hardware, we think that our solution will work out-of-the-box. The scheduler already uses up-to-date measurements regarding EPC usage to come up with scheduling decisions. Even when using SGX 1, variations of EPC usage can already happen if a container launches multiple enclaves whose life-cycles are not harmonized. The only part of our system that we have identified as not yet SGX 2-ready is our implementation of resource usage limits in the Intel SGX driver. We believe that the effort required to port it to the new revision of SGX is modest.

VII. Related Work

The problem of scheduling jobs over a cluster of heterogeneous machines has always attracted a lot of research.

To protect sensitive hypervisor scheduling decisions, *Scotch* [51] conveys information gathered in System Management Mode (SMM) to enclaves. The authors implement a prototype on top of the Xen hypervisor, adding about $14\,\%$ of overhead for each context switch and interrupt. Validation is done by checking the accuracy of accounting under resource interference- [52] and escape- [53] controlled attacks. Although they provide results within $2\,\%$ of the ground truth in such scenarios, there is no guarantee that measurements coming from SMM are not tampered with during control switch to enclave entry points. Their focus is on the protection of probing data and preventing improper resource usage. Instead, we deal with system support for scheduling SGX jobs based on their main contentious resource: EPC memory pages.

Similar to our work, *ConVGPU* [54] provides solutions for scheduling jobs based on memory contention in container-based virtualized environments. Specifically, they provide a mechanism that shares the GPU memory among multiple containers. Just as the EPC, GPU memory is limited. However,

it is not possible to swap out memory once it is full, an event that usually leads to more severe issues than just performance degradation. To avoid that, ConVGPU intercepts and keeps track of memory allocation calls to the CUDA API by providing an alternative shared library to applications running within containers. Whenever a request cannot be granted, it holds the application's execution by postponing the call return until there are available resources. They evaluate the system using four strategies for the selection of which waiting application should be served first, and show low overall application running time overheads. Essentially, they act reactively to potential memory contention issues, after container deployment, whereas we take scheduling decisions before, based on self-declared memory needs, and after deployment, based on probed data. Besides, they only take into account intra-node resource management, and leave distributed processing by integration with Docker Swarm for future work.

Checkpointing and migration of running processes closely relates to scheduling strategies. In this direction, Gu *et al*. [55] tackle these issues for SGX enclaves. The challenge lies in securely creating, transmitting and restoring an enclave checkpoint while preserving all security guarantees, while not introducing new attack vectors. Checkpointing a running SGX process, however, already imposes some obstacles, since part of the enclave metadata is not even accessible by the enclave itself. Moreover, for consistency reasons, one must ensure that all application threads do not continuously mutate the state of the job being migrated. The authors deal with the first issue by *inferring* the value of such unreadable metadata. Then, they issue replay operations that lead to an identical state. However, their approach rely on the cooperation of the untrusted operating system, and therefore check afterwards if it has behaved accordingly. The problem of achieving a *quiescent* point, when all threads are guaranteed to not modify the process state, is done by synchronization variables kept inside the enclave, and by intercepting its entry and exit points. By doing this, they force all threads to reach either a dormant or a spinning state that will only be undone after restoring the enclave at the target node. After successfully creating the checkpoint, they still have to provide means to ensure that it cannot be restored more than once (fork attack) nor that an old one can possibly be recovered (rollback attack). That is solved by means of a migration key transmitted through secure channels built by leveraging SGX attestation and by a self-destroy approach, which prevents the enclave from being resumed after it was checkpointed. Overall, authors [55] show a negligible performance overhead. Such mechanism could eventually be integrated into our system, towards a globally optimized EPC utilization through the migration of enclaves. However, we stress that the problem of SGX enclave migration (online or offline) is considered orthogonal to ours.

Similar to our work, *MixHeter* [56] also deals with scheduling in heterogeneous environments. Since different sorts of applications benefit from distinct hardware capabilities (*e.g*., GPUs for graph computing, RAM for sorting jobs, etc.), distributed system schedulers that deal with mixed workloads and take decisions disregarding this aspect may face poor performance. To that end, *MixHeter* proposes a scheduler based on *or-constraints*. The different resource requests are translated into algebraic expressions to be satisfied. If preferred resources are busy, non-preferred, but still compatible ones are used instead, which can maximize the overall performance. They evaluate the system on a popular scheduler for distributed systems processing frameworks and show performance improvements up to $60\,\%$. We only consider SGX and non-SGX jobs as characterizing features for orchestration decisions. Our system would therefore benefit from such a scheme considering a broader range of hardware capabilities, assuming applications would support alternative solutions (*e.g*., AMD Secure Encrypted Virtualization (SEV) [57], Trusted Platform Modules (TPMs) [58], ARM TrustZone [22]) in the absence of Intel SGX.

VIII. Conclusion

In this paper we have proposed a novel orchestrator for containers running on heterogeneous clusters of servers, with and without Intel Software Guard Extensions (SGX) support. This technology allows users to deploy their software data in the cloud without having to trust the providers. In SGX enclaves, software runs at almost native speed, unlike with cryptographic mechanisms that have severe limitations in terms of performance and features.

The challenge in such deployments is to schedule containers with security requirements to SGX machines in priority, which are scarce, while at the same time carefully monitoring their usage of SGX resources. In particular, when exceeding the limited memory capacity of SGX enclaves, performance starts degrading significantly so it is therefore important not to overload SGX machines with too many resource-demanding containers. To ensure proper monitoring of low-level SGX metrics, we extended the SGX Linux driver to gather statistics about the SGX runtime and feed them into the orchestrator, based on Kubernetes. We developed a complete prototype that we openly release [20], deployed it in a private cluster, and conducted a detailed evaluation using Google Borg traces. Our findings reveal that the scheduler must carefully take the Enclave Page Cache (EPC) size into account to reduce the overall turnaround time. Also, we observed that when half of the jobs in the workload are SGX-enabled, there is virtually no impact on general performance. Finally, our experiments show that there is a small bootstrap time that SGX containers must be ready to tolerate at startup.

As part of future research directions, we plan to extend our orchestrator by integrating support for enclave migration as well as hybrid processes running trusted and untrusted code.

Acknowledgment

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme and was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under grant agreement No 690111.

References

[1] Google, *Docker popularity trend over the last 5 years*, 2017. [Online]. Available: https://trends.google.com/trends/explore?date=today%205-y&q=docker.

[2] C. Anderson, "Docker," *IEEE Software*, vol. 32, no. 3, pp. 102–105, May 2015. DOI: 10.1109/MS.2015.62.

[3] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[4] Docker, *Docker Hub*, 2017. [Online]. Available: https://hub.docker.com/.

[5] Google, *Google Container Registry*, 2017. [Online]. Available: https://cloud.google.com/container-registry/.

[6] Redis, *Redis Docker Hub image*, 2017. [Online]. Available: https://hub.docker.com/_/redis/.

[7] MySQL, *MySQL Docker Hub image*, 2017. [Online]. Available: https://hub.docker.com/_/mysql/.

[8] Consul, *Consul Docker Hub image*, 2017. [Online]. Available: https://hub.docker.com/_/consul/.

[9] Google, *Google Cloud Computing Containers Service*, 2017. [Online]. Available: https://cloud.google.com/compute/docs/containers/.

[10] Amazon, *Amazon EC2 Container Service*, 2017. [Online]. Available: https://aws.amazon.com/documentation/ecs/.

[11] ClusterHQ, *ClusterHQ container orchestrators*, 2016. [Online]. Available: https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf.

[12] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "ContainerLeaks: Emerging security threats of information leakages in container clouds," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2017, pp. 237–248. DOI: 10.1109/DSN.2017.49.

[13] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.

[14] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '09, Bethesda, MD, USA: ACM, 2009, pp. 169–178. DOI: 10.1145/1536414.1536440.

[15] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *EUROCRYPT*, Springer, vol. 6632, 2011, pp. 129–148.

[16] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, p. 86, 2016.

[17] E. Rotem and S. P. Engineer, "Intel architecture, code name Skylake deep dive: A new architecture to manage power performance and energy efficiency," in *Intel Developer Forum*, 2015.

[18] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI'16, Savannah, GA, USA, 2016.

[19] M. Russinovich, *Introducing azure confidential computing*, Microsoft Azure, Sep. 14, 2017. [Online]. Available: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[20] S. Vaucher and R. Pires, *SGX-aware container orchestrator*, Source code. [Online]. Available: https://github.com/sebva/sgx-orchestrator.

[21] J. Wilkes, *More Google cluster data*, Google research blog, Nov. 2011. [Online]. Available: http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[22] ARM Limited, "Building a secure system using TrustZone technology," 2009.

[23] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "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*, ser. HASP 2016, Seoul, Republic of Korea: ACM, 2016, 10:1–10:9. DOI: 10.1145/2948618.2954331.

[24] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[25] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, 2017, pp. 645–658.

[26] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1816–1827, Aug. 2015. DOI: 10.14778/2824032.2824078.

[27] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," *SIGMOD Rec.*, vol. 23, no. 2, pp. 419–429, May 1994. DOI: 10.1145/191843.191925.

[28] Y.-W. Huang and P. S. Yu, "Adaptive query processing for time-series data," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 1999, pp. 282–286.

[29] Kubernetes, *Kubernetes components*, 2017. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[30] ——, *Device plugins*, 2017. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/device-plugins/.

[31] *gRPC: A high performance, open-source universal RPC framework*, 2017. [Online]. Available: https://grpc.io/.

[32] Kubernetes, *Heapster*, 2017. [Online]. Available: https://github.com/kubernetes/heapster.

[33] influxdata, *InfluxDB*. [Online]. Available: https://www.influxdata.com/time-series-platform/influxdb.

[34] ——, *InfluxQL*, 2017. [Online]. Available: https://docs.influxdata.com/influxdb/v1.3/query_language/.

[35] Kubernetes, *Kubernetes DeamonSet*, 2017. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

[36] P. Menage, P. Jackson, and C. Lameter, *Cgroups*, 2004. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[37] A. Gerrand, *C? Go? Cgo!* The Go Blog, Mar. 17, 2011. [Online]. Available: https://blog.golang.org/c-go-cgo.

[38] *ioctl(2), Linux programmer's manual*, May 3, 2017. [Online]. Available: http://man7.org/linux/man-pages/man2/ioctl.2.html.

[39] Intel Corporation, *Intel SGX Linux driver*, commit `eb61a95`. [Online]. Available: https://github.com/01org/linux-sgx-driver.

[40] S. Vaucher, R. Pires, and C. Fetzer, *Modifed Intel SGX Linux driver*, commit `600e3b2`. [Online]. Available: https://github.com/sebva/linux-sgx-driver.

[41] Intel Corporation, *Intel Software Guard Extensions SDK for Linux OS, Installation guide*, version 2.0, Nov. 2017. [Online]. Available: https://download.01.org/intel-sgx/linux-2.0/docs/Intel_SGX_Installation_Guide_Linux_2.0_Open_Source.pdf.

[42] S. Vaucher, *SGX-enabled Docker image*, Docker Hub. [Online]. Available: https://hub.docker.com/r/sebvaucher/sgx-base/.

[43] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, Revised 2012.03.20. [Online]. Available: http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[44] S. Vaucher, V. Schiavoni, and P. Felber, "Stress-SGX: Load and stress your enclaves for fun and profit," in *Networked Systems*, Cham: Springer International Publishing, 2018.

[45] C. I. King, *Stress-ng*. [Online]. Available: http://kernel.ubuntu.com/~cking/stress-ng.

[46] R. Pires, M. Pasin, P. Felber, and C. Fetzer, "Secure content-based routing using Intel Software Guard Extensions," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16, Trento, Italy: ACM, 2016, 10:1–10:10. DOI: 10.1145/2988336.2988346.

[47] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "Securekeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16, Trento, Italy: ACM, 2016, 14:1–14:13. DOI: 10.1145/2988336.2988350.

[48] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, 2017.

[49] T. Bui, "Analysis of Docker security," *CoRR*, vol. abs/1501.02967, Jan. 2015. arXiv: 1501.02967.

[50] Intel Corporation, *Intel 64 and IA-32 architectures software developer's manual*, 4 vols. Oct. 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm.

[51] K. Leach, F. Zhang, and W. Weimer, "Scotch: Combining Software Guard Extensions and system management mode to monitor cloud resource usage," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, Springer, 2017, pp. 403–424.

[52] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, Raleigh, North Carolina, USA: ACM, 2012, pp. 281–292. DOI: 10.1145/2382196.2382228.

[53] K. Kortchinsky, "CLOUDBURST: A VMware guest to host escape story," in *Black Hat*, 2009.

[54] D. Kang, T. J. Jun, D. Kim, J. Kim, and D. Kim, "ConVGPU: GPU management middleware in container based virtualized environment," in *IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 301–309.

[55] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure live migration of SGX enclaves on untrusted cloud," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2017, pp. 225–236. DOI: 10.1109/DSN.2017.37.

[56] X. Zhang, Y. Lyu, Y. Wu, and C. Zhao, "MixHeter: A global scheduler for mixed workloads in heterogeneous environments," *Journal of Parallel and Distributed Computing*, vol. 111, no. Supplement C, pp. 93–103, 2018. DOI: 10.1016/j.jpdc.2017.07.007.

[57] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," Advanced Micro Devices, White Paper, 2016. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[58] Trusted Computing Group, *Trusted computing platform alliance, Main specification*, version 1.1b, 2002. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TCPA_Main_TCG_Architecture_v1_1b.pdf.