Clemmys: Towards Secure Remote Execution in FaaS

Bohdan Trach[†], Oleksii Oleksenko[†], Franz Gregor[†],
Pramod Bhatotia[‡], Christof Fetzer[†]
[†]Technische Universität Dresden [‡]The University of Edinburgh

Abstract

We introduce Clemmys, a security-first serverless platform that ensures confidentiality and integrity of users' functions and data as they are processed on untrusted cloud premises, while keeping the cost of protection low. We provide a design for hardening FaaS platforms with Intel SGX—a hardware-based shielded execution technology. We explain the protocol that our system uses to ensure confidentiality and integrity of data, and integrity of function chains. To overcome performance and latency issues that are inherent in SGX applications, we apply several SGX-specific optimizations to the runtime system: we use SGXv2 to speed up the enclave startup and perform batch EPC augmentation. To evaluate our approach, we implement our design over Apache OpenWhisk, a popular serverless platform. Lastly, we show that Clemmys achieved same throughput and similar latency as native Apache OpenWhisk, while allowing it to withstand several new attack vectors.

CCS Concepts

• **Security and privacy → Distributed systems security**;

ACM Reference Format:
Bohdan Trach[†], Oleksii Oleksenko[†], Franz Gregor[†],, Pramod Bhatotia[‡], Christof Fetzer[†]. 2019. Clemmys: Towards Secure Remote Execution in FaaS. In *The 12th ACM International Systems and Storage Conference (SYSTOR '19), June 3–5, 2019, Haifa, Israel.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3319647.3325835

1 Introduction

Serverless Computing. Serverless computing, or Function-as-a-Service (FaaS), is a cloud computing paradigm that emerged to make processing of bursty, irregular event-driven workloads cheaper, and deployment—simpler [12]. To reap these benefits, application developers must decompose their software in terms of the core abstraction of FaaS—a *function*: a short-lived single-purpose service that is spawned to process a single event.

Serverless paradigm implies processing data with the stateless functions, using a fresh container to serve each request or event. From the programmer's point of view, functions are written to an API and a set of libraries specified by the platform owner, and without any assumptions about the persistence of local files or about the underlying hardware. This concept is already implemented in multiple open frameworks [1, 8–10] and commercial platforms [3, 5, 6]).

Serverless computing runs with the promise of *perfect scalability*: most requests are resource-intensive, but cloud provider performs optimal function placement across the necessary number of nodes. Such common tasks as web page serving do not need this guarantee, since they are I/O bound and run for milli- or microseconds. Yet, this promise is vital for running workloads that are CPU-bound and run for dozens of seconds in the cloud, as, otherwise, developers would be forced to scale the system without insights into resource availability. Thus, programmers are freed from implementing load-balancing and autoscaling solutions for their services, further reducing development cost.

The benefits that serverless computing presents to the users are twofold. First, the user is freed from making decisions about platform management, security, and software updates. These tasks are delegated to the cloud provider, who can handle them using the available infrastructure knowledge. Second, with short-lived services billed per invocation, it is possible to run in an economically efficient way even those services that are idle most of the time.

Trust issues. Despite the significant economic benefits that come with serverless computing, the trust issues could become a dealbreaker when it comes to processing sensitive data in the cloud. Specifically, there are two main aspects that make cloud services extremely challenging to secure.

First, cloud applications run with a large set of system components, which must function correctly for the system to maintain its security properties, that is, with an excessive Trusted Computing Base (TCB). In the cloud, TCB includes the host (operating system, hypervisor), and the userspace stack running on the machine. Due to the large size and complexity of these components, they are likely to have

SYSTOR '19, June 3–5, 2019, Haifa, Israel
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325835

flaws, and it is probable that attackers can find exploitable vulnerabilities and subvert the platform security properties.

Second, as trust in the cloud provider is unavoidable, outsourcing to the cloud implies giving the provider's employees access to the users' data. If the employees have malicious intentions, it could have grave consequences, especially in case of medical and financial applications.

Although these issues are becoming a major obstacle to adoption of serverless computing, no platform currently tries to solve them in a principled way.

Trusted Execution Environments. There are several potential ways to approach the trust problems. If it is necessary to only store the data, users can rely on end-to-end encryption to ensure safety. However, in most cases, cloud applications need to also process the data, which is not possible with end-to-end encryption. Applications could employ fully homomorphic encryption schemes to enable modification of the encrypted data [11], but the currently available schemes are too restrictive and too costly to be used in practice [35].

Another approach is to use Trusted Execution Environments (TEE) [2, 15, 25]. TEEs provide isolated memory regions for code and data, which are not accessible by the privileged software running on a host. Thus, developer can use these regions to store the data that must stay confidential and integrity-protected, that is, modified only by the code stored inside this region. TEEs cause much lower overhead compared to homomorphic encryption. Therefore, we chose them as a basis of our work.

Clemmys. In this paper, we present a system that allows users to benefit from serverless computing while preserving security of their data, for which we rely on the protection provided by TEEs.

To tackle this problem without incurring prohibitive overheads, only two components of Clemmys are running inside a TEE—the platform gateway and the user functions. Additionally, we develop a message format that preserves message confidentiality and integrity while the data is sent between the other, unprotected platform components.

Our contributions include:

- We design and implement a reencrypting proxy that terminates TLS connections and encodes function invocations into our message format.
- We develop a message format that preserves message confidentiality and integrity and can be used in a variety of FaaS platforms.
- To prove the validity of our approach, we implement Clemmys in OpenWhisk, a popular FaaS platform.
- To reduce the cost of protection, we additionally implement several important SGX-specific optimizations.

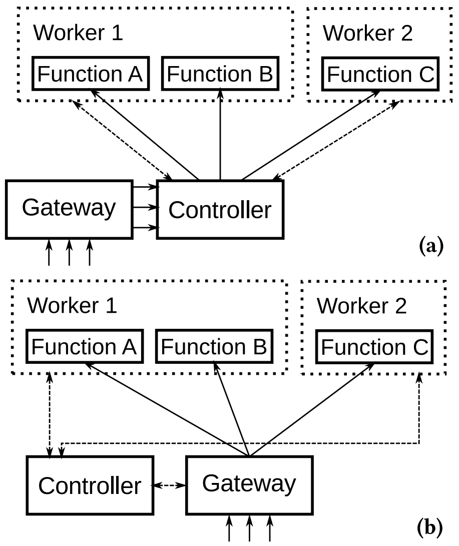

Figure 1: FaaS platform architectures: (a) with controller node as load balancer; (b) with gateway as load balancer.

2 Background

Serverless Computing Platforms. To design a usable and generic defence strategy for common serverless platforms, we begin by analyzing similarities among them. For this, we have studied architectures of several open serverless platforms: OpenLambda, Iron.io, OpenWhisk, Fission.io.

We found out that all of these architectures can be boiled down to two types shown on Figures 1a and 1b. The architectures contain a gateway node, a controller node, and multiple worker nodes. The architecture may also contain service nodes such as a database system to store platform configuration, message queues, and so on (not shown in the figures).

The *worker* nodes execute functions with specified inputs and resource limits in response to network events. In production deployment, there will be numerous worker nodes, necessary to provide the advertised level of scaling.

The *controller* node is the central management component of the platform. It manages the available functions images, resource limits, user accounts, billing, and permissions. Typically there is a single controller instance connected to a database for metadata storage.

To ensure secure connection of the platform to the Internet, it uses a *gateway* service that terminates the TLS connections from client and acts as a load balancer to the workers. Some systems (e.g., OpenWhisk) use the controller as a load balancer.

A connection between the gateway and a function can be either direct or indirect. For example, a design may include a message bus for reliability, so that messages that were admitted to the system are guaranteed to produce a result.

A common pattern in serverless computing is *chaining*—composition of functions into sequences where data is passed from function to function without involvement of the user. The functions in the chain can run either on different nodes, or on a single node (for data locality).

Intel SGX and Scone. Intel SGX is an extension to the Intel X86 architecture that allows users to create *enclaves*—hardware-isolated memory ranges, with code inside enclave running in a special *enclave* execution mode. In this mode, memory reads that originate from code outside of the enclave cannot read the enclave memory, and attempts to jump to the code outside of the enclave fail. These semantics protect from attacks by privileged software, as they prevent the operating system, hypervisor, and SMM (System Management Mode) from accessing enclave memory. The physical memory that backs the enclave (called Enclave Page Cache, *EPC*) is encrypted by hardware. SGX also has remote attestation capability verifies that the application is indeed running inside the enclave.

SGX has a number of limitations. Enclave entries and exits have a very high cost: these events cause costly TLB and cache flushes, imposing a significant cost on both enclaved and colocated software. Also, the EPC size in the currently available CPUs is limited to 128Mb, out of which ~98Mb is available to the software. When the amount of virtual enclave memory exceeds the amount of EPC available, *EPC paging* takes place—stale EPC pages are encrypted and written out to unprotected memory, while the requested ones are put back into the enclave. EPC paging has a significant performance overhead and is a major problem for SGX applications.

Recent Intel CPUs support the second generation SGX (SGXv2). The main addition in SGXv2 is support for Enclave Dynamic Memory Management [24], which allows adding (*augmenting*), modifying metadata, and removing (*trimming*) EPC memory from the enclave after it started running. We use SGXv2 to lift some of the SGXv1 restrictions.

In this work, we build on Scone shielded execution framework, which is a framework for running unmodified POSIX applications on top of Intel SGX enclaves [31].

Palaemon. Palaemon[1] is a key management service (KMS) implemented as a part of the Scone remote attestation and configuration system, which uses an attestation scheme similar to that of ShieldBox [19, 33]. It supports standard KMS features, such as a flexible policy language for specifying secrets and entities that may access them and automatic generation of secrets. Most importantly, it supports provisioning secrets and shielding layer keys to Scone-based applications.

Palaemon is implemented to run inside an Intel SGX enclave alongside with the applications it attests; Palaemon itself is attested using Intel Attestation Service (IAS) [19]. It opens a possibility to operate Palaemon as a turn-key solution, that is using a single instance per data center.

Palaemon consists of two components: Local Attestation Service (LAS) and Configuration and Attestation Service (CAS). LAS is running on the same node as the attested application, and issues SGX local attestation quotes to the CAS. LAS itself is attested using IAS. CAS is the service that securely issues the configuration to the correctly attested applications.

In our work, this configuration comprises function chain configuration, cryptographic keys, and function-specific information.

3 Threat Model

We consider a typical scenario of FaaS platform operation. A *user* acquires a function source from a *function provider*. The function is deployed on a cloud platform managed by an *operator*, who has access to the host OS on all nodes. A malicious *external attacker* may try to exploit a vulnerability anywhere in the function or in the cloud stack to gain access to the function source or data. This scenario is the foundation of our threat model:

- The operator should not be able to compromise confidentiality and integrity of the function source and data.
- The function provider should not be able to compromise confidentiality and integrity of the data processed by the function.
- Only the user should be able to define the function chain contents. Other parties should not be able to drop or add arbitrary function to the user's chain.

Specifically, we target the following attack vectors:

AV1. The operator or the external attacker inspect the memory of the functions or of the system components. This way, they can extract session keys and decrypt the traffic, or directly extract plaintext user data from the process memory.

AV2. The operator reads and modifies the traffic between the function and its users. This is possible because messages between gateway and the functions are unencrypted. The external attacker could also intercept the traffic if she compromises a part of the cloud stack.

AV3. The operator modifies the execution order of functions in a chain to create an information disclosure. For example, the message content that is supposed to be sent to the user can be redirected to a logger that writes or sends plaintext messages over the network.

In our threat model, we do not consider microarchitectural attacks or memory safety [27] attacks. We assume that approaches like Cloak [20], Varys [29], or SGXBounds [23] will thwart them.

We also consider application vulnerabilities as orthogonal to our work. To prevent leaks of information due to neglect

[1]The paper that describes Palaemon in detail is currently under submission.

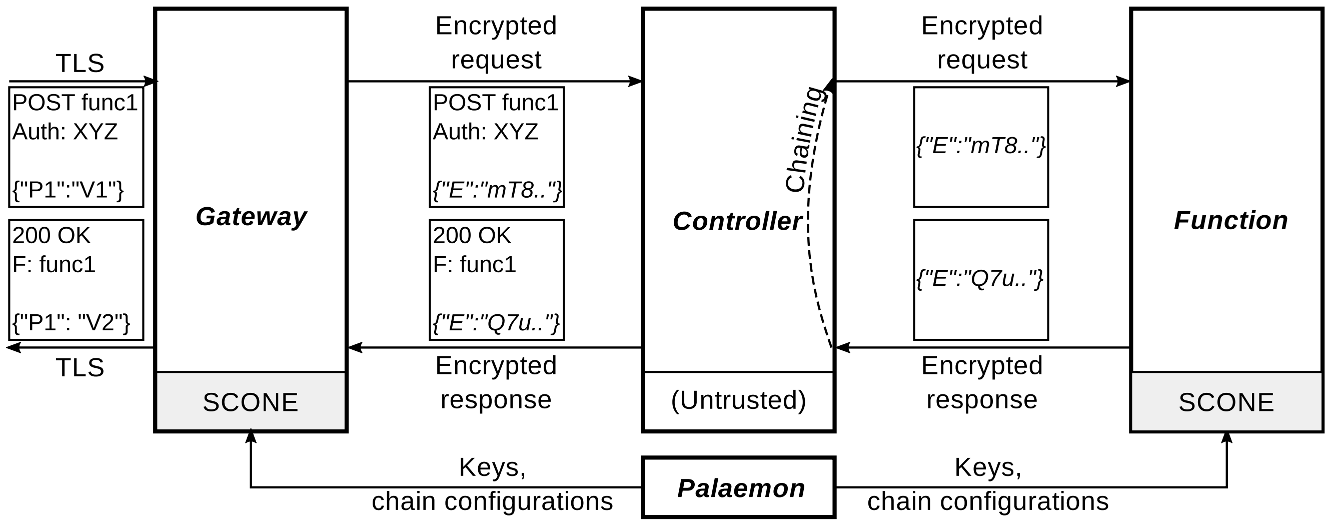

Figure 2: System architecture of Clemmys and transformations of a user request as it passes through the system.

or malicious actions of the function provider, the user must manually inspect the function code. While theoretically a form of static analysis or sandboxing can prevent such leaks, we do not tackle this problem.

As with Intel SGX system resource allocation remains under control of privileged software, Clemmys does not guarantee availability.

4 Design

In this section, we discuss how we can secure common FaaS platforms from the attacks outlined in the threat model (§3). We defend against AV1 by running each function inside Intel SGX enclaves. We prevent AV2 by encrypting traffic between the function and the gateway. To tackle AV3, we introduce a protocol that cryptographically ensures the correct order of functions.

Combined, Clemmys comprises the following generic system architecture (see Figure 2). First, the user initiates a mutually authenticated TLS connection to the gateway. Then, every incoming HTTP request passes through the gateway which terminates the TLS connection, reencrypts the message body into an internal message format, and passes it to the FaaS controller. The controller inspects the request metadata (not modified by the gateway), and passes it to the appropriate function on the target machine, possibly via a message queue. On the target machine, the container is started to process the message. As an optimization, the platform could reuse a container from a previous request. The function starts inside a Scone SGX enclave and performs remote attestation and configuration using Palaemon. Then, the function verifies that it is executing at the right stage of the chain using the information from Palaemon and from the message, decrypts and processes it. When the result is ready, the function encrypts it and passes it either to the controller or to the next function, depending on the current stage of the chain. When the final result is computed, the gateway decrypts it and writes it down to the client TLS connection.

4.1 Preventing Memory Inspection

Clemmys targets **AV1** by employing SGX enclaves that hide the memory contents from the adversaries and ensure its integrity and confidentiality. Of all the system components, only the API Gateway and the functions have to run in SGX containers as these components are the only ones that interact with raw user data: The rest of the system deals with benign metadata, and the message content is encrypted (§4.3).

4.2 Preventing Traffic Analysis and Modification

We prevent **AV2** by introducing encryption between the gateway and the functions. Note that for FaaS platforms like OpenWhisk we cannot use TLS between the gateway and functions, and between several functions, as a message queue may be deployed there for reliability. Message queues are typically used in cloud settings to decouple services and to gain message persistence.

At this point, two alternatives are available. The strawman solution is to run all system components inside Intel SGX enclaves and extend the components to use TLS for communication. However, this solutions comes at a significant cost in case serverless platform has memory-intensive components, for example Kafka. To secure these components, it would be necessary to use SGX enclaves, where this software would experience slowdown due to EPC paging [31]. Another alternative is to encrypt the messages at the first client-facing node into format which can be passed through the system transparently. We explain this approach in §4.3, where we tackle AV2 and AV3 together.

4.3 Verifying Function Execution Order

We target **AV3** by designing a protocol which cryptographically certifies that the functions are invoked in the order specified by the developer. Our key observation is that the user data in the messages are not read by the intermediate nodes in any way; instead, the nodes rely on metadata transmitted in, for example, HTTP headers to schedule the execution of functions. Thus, it is possible to encrypt the message data and add extra information to it without any effects on the system operation.

To facilitate secure function chaining, the protocol must allow functions to detect and to cryptographically verify the following violations of function chaining:

- A function is dropped from the chain.
- A function is inserted into the chain.
- The functions are executed in a wrong order.

We recognize that the message of the protocol should contain the plaintext metadata (user certificate fingerprint and function chain name) to query the decryption key from Palaemon. Additionally, the format must include information necessary to identify if the processing happened in the right order. To achieve this goal, we store the chain (list of functions) inside Palaemon, and include the index of the functions in the chain in the protocol message.

Thus, given a message[2] M, the resulting protocol message has the following fields:

$$BASE_{64}(C,CN,N,IV,AESGCM(M,IV,\langle C,CN,N\rangle)_K) \qquad (1)$$

where:

- C—fingerprint of user certificate;
- CN—name of the function chain (carries no semantic information for the function);
- N—index of the current function in the chain;
- $AESGCM(M,IV,B)_K$—AES-GCM encryption of plaintext message M using key K, initialization vector IV, and associated data B;

The function can use the certificate fingerprint C, chain name CN, and index N to detect the violations as follows. First, it performs remote attestation, and gets lists of functions in the chain and AES-GCM key for each C, CN pair from Palaemon. Then, it uses the fingerprint and function chain name to select the correct AES-GCM key, and verify that the attacker did not modify the abovementioned fields. The action uses the function name and index fields to ensure that the processing in chains happens in the specified order. Action looks up a function with index N in the chain CN, and checks if it matches the identity of the action currently executing. If there is a match, execution of the function chain is correct. The index field N is incremented as function finishes execution and passes the message to the next function for processing.

[2]The message is in JSON as native OpenWhisk uses this encoding to exchange information among the system components.

Function Identity. To allow chain verification, an action running with SGX must be able to learn and verify its name (i.e. *identity*) CN. In the simple case when the identity corresponds to a single binary, we can ensure its validity using information from Scone and Palaemon. When Scone builds an enclave, it also calculates a cryptographic checksum of its initial image, called enclave measurement, verified during the remote attestation. Palaemon can send an attested enclave a secret that depends on the enclave measurement. In our case, Palaemon sends the enclave the intended identity for this measurement after the remote attestation.

However, actions implemented in interpreted programming languages require additional care. As the attestation verifies only the interpreter and the libraries, the interpreted application source is not included in the attestation report. Thus, for such functions, additional measures are necessary to bind the function identity to the executing memory image. For Python, we suggest using Scone file system shield with integrity and confidentiality protection [31]. In this scenario, Palaemon performs remote attestation of the Python interpreter and sends the keys to the enclave for decrypting the shielded file system image. Then, Python can read the identity of the function from the file in the shielded file system. Support for this identity verification requires changes to the Python interpreter, to prevent it from loading executable scripts from unprotected file systems.

Key and Configuration Management.. The protocol message outlined in §4.3 uses symmetric AES-GCM encryption. The keys used for the encryption are generated and stored in an external key management service—Palaemon—with a unique key for every user-chain pair. It allows Clemmys to isolate requests from different clients and prohibits the adversaries from moving a message from one chain to another.

Palaemon also stores information necessary for the chain integrity protocol:

- Lists of functions inside each chain in the system.
- Bindings of function names to enclave measurements (for native functions, C/C++/Rust).
- Decryption keys to function sources stored in a protected file system (for interpreted programming languages, Python/Node.js).

The protected components (gateway and the functions) fetch the keys and other configuration data at the startup of the corresponding program, after the remote attestation. Thus, an extra round trip is required to start the application. Its effect, however, is moderate as all communication with Palaemon happens locally, unlike during the Intel attestation procedure.

Clemmys relies on client certificates for client authentication, instead of the authentication system available in the FaaS platform. Clients generate a key and use it to receive a valid certificate signed by the Clemmys key. The gateway verifies the client certificates and rejects connections with those clients that do not present one. The fingerprint of the presented certificate is used in the protocol message to identify the client. We make this decision because FaaS controller is run without SGX protection in our case, and therefore can be easily attacked by a privileged adversary.

5 Implementation

We rely on Apache Openwhisk to implement Clemmys. To keep the performance impact of the protection low, we strived to restrict the changes to as few components as possible.

API Gateway. In the original OpenWhisk design, API Gateway terminates TLS connections and manages functions triggered over REST. The original API Gateway is implemented using OpenResty (Nginx distribution with Luajit and numerous Lua extensions). Therefore, to extend the API Gateway with the message reencryption functionality (as explained in §4.3), we have implemented a dedicated Nginx plugin.

The core functionality of our plugin is:

- Maintaining key and chain information after startup and remote attestation.
- Performing message reencryption.
- Performing security checks on the messages and client connections.

The plugin implements an Nginx rewrite phase handler to encrypt the request before passing it to the OpenWhisk Controller, and uses the header and body filter to receive the reply, verify its correctness, decrypt it, and pass it to the client (1230 lines of C code in total). We use the Nginx configuration file to specify the REST endpoints for which the plugin must be active.

At the plugin initialization, it scans through the process environment variables supplied by Palaemon, and populates the configuration tables, which are later used to process user requests. So far, Palaemon does not support dynamic updates to the configuration: to receive new chain configurations and keys, the API Gateway has to be restarted. Alternatively, operators can use systems like HAProxy [7] to perform a zero-downtime configuration update. We plan to alleviate this limitation of Palaemon and the Nginx plugin in the future.

Function image skeleton. We implement our own skeleton images for native SGX functions and Python SGX function. The native SGX image adds a configuration file for Scone asynchronous system call interface and configures the environment variables to set default enclave heap size. The SGX Python image additionally replaces the stock Python from the Alpine Linux repository with Scone-build Python and installs a set of predefined Python packages using pip utility and Scone cross-compiler.

Controller. We modified Controller to put the action name in the header of replies to the "activation get" request. This change allowed us to avoid costly parsing that we would have to do to secure this REST endpoint.

Invoker. Invoker is a service running on the worker node that communicates with the Kafka queue and launches containers with functions to serve the user requests. The input from the user is provided to the function via standard input.

We have also modified Invoker to pass additional SGX-specific parameters to the Docker. Our modified Invoker adds `/dev/isgx` device to the function container. Also, it configures additional container resource limits required by Scone.

5.1 Function Startup Optimization

The requirement of not degrading the system latency clashes with the reality of running dynamic language runtimes inside SGX enclaves. These runtimes typically run with huge heaps, which increases the startup time of SGXv1 to the range of seconds, or even tens of seconds (Table 1). To a large extent, it is caused by the enclave loading procedure.

The enclave loading procedure has several steps: creating control structures, adding and measuring EPC pages, and launching the enclave using EINITTOKEN structure. Based on our observations, the main bottleneck is in adding pages: Even though heap pages are not measured, adding them to the enclave still takes significant amounts of time. To reduce the attack surface, the Scone mmap implementation also zeroes the added pages, which slows down the application startup when large chunks of memory are allocated.

To mitigate the impact of this issue, we rely on SGXv2 EDMM [24] features to reduce the startup time. We use the SGXv2 EPC augmentation feature to skip adding most of the pages during enclave creation, which can take a significant amount of time during the enclave startup, especially with large heaps. Instead, we allocate only a small number of heap pages at the beginning of the heap region—20 MB by default—and around 40 pages (164 kB) at the end of the region, where the bitmap for mmap allocator is located. Scone loader skips adding all other pages. These changes sum up to significant savings: the SGX loader issues an `ioctl` to the driver for each page that needs to be added, which copies the full page contents into the kernel before adding them to the enclave. Avoiding this work brings a significant improvement to the startup time. Batching cannot reduce this cost, because it is mostly caused by EPC metadata updates, not `ioctl` calls.

On top of the SGXv2 support, we implement two additional optimizations: batch EPC augmentation and zeroing pages upon deallocations instead of allocations.

Because most of the accesses during startup cause augmenting enclave exits, we modify the driver and Scone runtime to do batch augmentation of enclave memory. On the EPC augmentation event, we prefetch a block of B pages containing the faulting address. We allow users to configure the amount of batching, as this optimization may not have a large effect on most applications. While this optimization moves cost of adding pages to time after application start, it distributes the faulting memory accesses over long time, reducing the EPC paging rate.

We also modify Scone to perform mmap page zeroing on page deallocation, instead of page allocation[3]. The benefits of this design are twofold. First, the fresh pages added to the enclave via the augmentation mechanism are automatically zeroed by the hardware. Thus, enclaves can use these pages without zeroing them. Second, the application may optimize its run time by exiting without zeroing memory

In general, we expect a much lower effect from the latter two optimizations than from just switching to SGXv2. Our optimizations are orthogonal to those published in recent studies [13, 26] and can be applied independently.

6 Evaluation

In this section, we answer the following questions:

- How does Clemmys perform compared to native OpenWhisk in terms of the best achievable throughput and latency?
- What is the performance impact of the function chain integrity protocol?
- What is the impact of our optimizations on the function startup latency?

Applications. We base our evaluation on the Fex [28] evaluation framework, with six computationally-intensive applications from PARSEC [18] and Phoenix [30] benchmark suites as workloads. We used the largest inputs that do not cause intensive EPC paging.

Methodology. The reported results are averaged over 10 runs and "mean" is a geomean across all the benchmarks.

Testbed. We ran all the experiments on two machines with different generations of the SGX technology. The machine with SGXv1 has a 4-core (8 hyperthreads) Intel Xeon CPU operating at 3.6 GHz (Skylake microarchitecture) with 32 KB L1 and 256 KB L2 private caches, an 8 MB L3 shared cache, and 64 GB of RAM. The machine with SGXv2 is a NUC with a 4-core (4 hyperthreads) Intel Pentium Silver CPU operating at 1.5 GHz (Apollo Lake microarchitecture) with 16 KB L1 and 128 KB L2 private caches, an 4 MB L3 shared cache, and 32 GB of RAM[4].

[3]Note that deallocated pages are kept within EPC, and thus, the OS cannot modify their contents in-between allocations.

[4]This was the only released SGXv2-enabled machine at the submission time.

6.1 Security Evaluation

We begin with a security evaluation of the protocol outlined in the §4.3. To this end, we run a chain of two echo functions implemented in Python with different identities. When the message processing order corresponds to the specified processing order in the chain, the functions succeed. Then, we modify several components of the message and test if the modified message is accepted. The results were as follows.

- Changes to certificate fingerprint, chain name, and counter are detected by AES-GCM tag comparison.
- Sending the message to a wrong function in a chain causes function failure due to mismatch between function identity and the intended function in the chain.

Finally, to simulate a rollback attack, we save the message after it has been processed by the first function, and separately invoke the second function with the same message. We see that the second function accepts and processes this message without signalling an error, even though this message has been already processed before. Thus, we can see that Clemmys offers no defence against rollback attacks. This attack vector targets only stateful functions, which must counter it at application level.

6.2 Response time

Next, we measure the impact of Clemmys on the overall system response time. Because Clemmys hardens two components of OpenWhisk—API Gateway and the functions—we perform two experiments to evaluate their impacts separately. In the first experiment, we evaluate the impact of the Gateway in isolation by running native functions with two configurations of the Gateway: native (as in OpenWhisk) and protected (with our Nginx plugin, see §5). In the second experiment, we isolate the impact of protecting the functions by running a protected Gateway with native and SGX-protected functions. In both cases, we use SGXv2 machine as the function startup time on it is is independent from heap size (see §6.3 for details). The response times were measured for different levels of oversubscription, from 1 to 16 instances of a function running in parallel. This measurement shows how many enclaves running in parallel the system can sustain on a single worker node. This information allows operators to assess the requirements to their hardware platform.

The measurements of the Gateway and function impacts are presented, accordingly on Figures 3 and 4. As we see, the cost of protection is low and mainly amortized by the overhead of OpenWhisk itself. Note however, that the memory consumption of the workloads was relatively low and did not cause intensive EPC paging.

6.3 Function startup optimizations

To evaluate the impact of our SGXv2-based optimizations on the function startup time, we do measurements on the SGXv2 machine. The Scone configuration uses a single enclave and

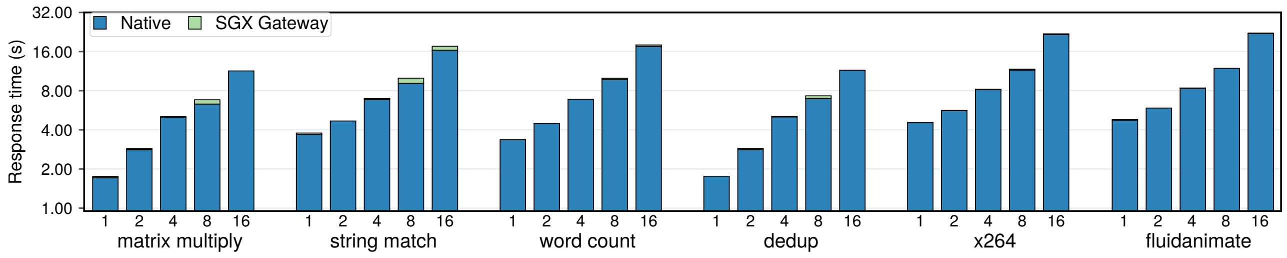

Figure 3: Response time with a protected API Gateway compared to native API Gateway, for different numbers of functions sharing a machine. The functions are non-protected to highlight the impact of the Gateway.

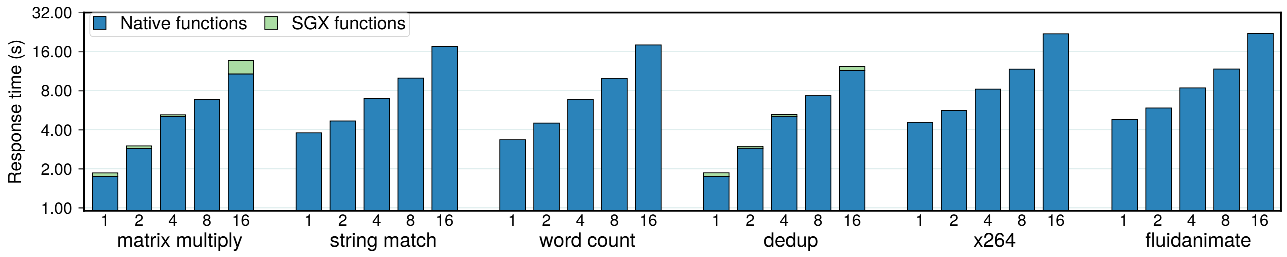

Figure 4: Response time with CLEMMYS-protected functions compared to native functions, for different numbers of functions sharing a machine.

a system call thread, with 4Gb heap. We chose this particular heap size because it exemplifies a CPU-bounded application running with a dynamic programming language runtime.

We evaluate several versions of the runtime:

- SGXv1—no optimizations;
- SGXv2—SGXv2 version with EPC augmentation batch of 20 pages;
- SGXv2 (NB)—SGXv2 version with the batching disabled (No Batching);
- SGXv2 (NB, NO)—SGXv2 version without batching and without optimized mmap allocator (No Batching, No Optimizations).

Because of the greatly varying runtimes of experiments, we normalize the results to the runtime of SGXv1 version. We skipped raytrace benchmark in all cases as it required X11 libraries to build.

We show the results of these experiments on Figure 5. We can see that in all cases there is a significant speedup from using SGXv2. On average, applications have ~20× lower latency on Parsec benchmarks, and 10× lower latency on Phoenix benchmarks.

To explain these results, we conducted an additional experiment (Table 1), where we measured application startup time of a no-op C application while varying heap sizes with SGXv1 and SGXv2. We discovered that enclave startup time depends linearly on the heap size in SGXv1 case. With larger heap sizes initialization time can reach 35 seconds (for 4GB heap), and it dominates the total application runtime. For comparison, the initialization time for 4 GB heap with SGXv2 is ~0.37 seconds. Thus, the results on Figure 5 do not mean that the applications were running faster, only that the cost of enclave initialization became greatly reduced.

The impact of additional optimizations is limited compared to just switching to SGXv2. On short-running Phoenix benchmarks (figure not shown), these optimizations do not have any influence. After investigation of PARSEC benchmarks, we have discovered that only *dedup* and *facesim* benchmarks had working sets larger than the EPC size. All other benchmarks dynamically allocate less than 30 Mb of memory, experience no EPC paging and get no benefit from the proposed optimizations. The impact of the optimizations is smaller on the *dedup* benchmark: after the initial setup phase, *facesim* allocates memory without freeing it, leading to highly local memory use. On the other hand, *dedup* has approximately 1 free call per 2 allocations, reducing the memory allocation locality. Thus, we conclude that our optimizations are benefitial only for functions that constantly allocate memory, and have no influence on most functions.

6.4 Impact of API Gateway

We evaluate the API Gateway of our system to answer the following question: *At which point does the API Gateway become a bottleneck in our system compared to the native version?* To answer this question, we perform benchmarking using our modified API Gateway running inside SGX enclave, and a native version of API Gateway. The reencryption plugin is disabled in the native version, otherwise the configuration file used is the same in both version. We did not perform any

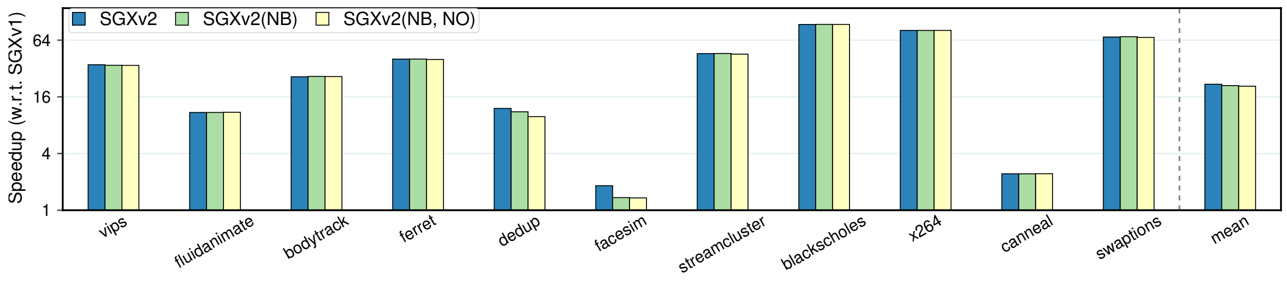

Figure 5: Parsec benchmarks results for our SGXv2-based optimizations.

Heap Size	SGXv1 startup time	SGXv2 startup time
4 Gb	35 s	0.37 s
2 Gb	15.6 s	0.37 s
128 Mb	0.84 s	0.37 s

Table 1: Startup time of a no-op enclave depending on the dedicated heap size for SGXv1 and SGXv2.

advanced Nginx configuration tuning. We perform evaluation in two scenarios:

- With dummy OpenWhisk functions: We run stock OpenWhisk with a minimal function after the API Gateway. The function is implemented in C and returns a hardcoded correct message.
- Without functions: Gateway passes the request to the Nginx upstream that replies to all requests with the same hardcoded message (OpenWhisk is not used).

We run the experiments on SGXv1 machine. We skip the second scenario with a native API Gateway, as it corresponds to normal file serving over HTTPS. On our machine, the saturation point for a single core is at 6000 requests/s.

The results are on Figure 6. We can see that without functions SGX-based API Gateways scales up to 300 requests/s. With the OpenWhisk functions, we see that both native and SGX versions saturate at around 225 requests/s, suggesting a bottleneck in the OpenWhisk components different from API Gateway. The increased latency at 25 requests per second is caused by the increased rate of container spawning, which increases system load; but this rate is insufficient to always cause container reuse: 14% of all containers at this rate are freshly started, vs. less then 1% at 100 requests/s. On the other hand, at the higher requests rates, starting at 200 requests/s, the system is running above its capacity and queuing excessive messages in the Kafka queue; the fresh container ratio at this rate is approximately 2%, and the (median, 95th percentile) latency at Invoker is (0,045s, 0.06s).

7 Discussion

As we mentioned in §1, typical FaaS applications are CPU-intensive, run for several seconds, and require scalability guarantee. Applications that are I/O bound or have very short runtime would have huge overheads from running

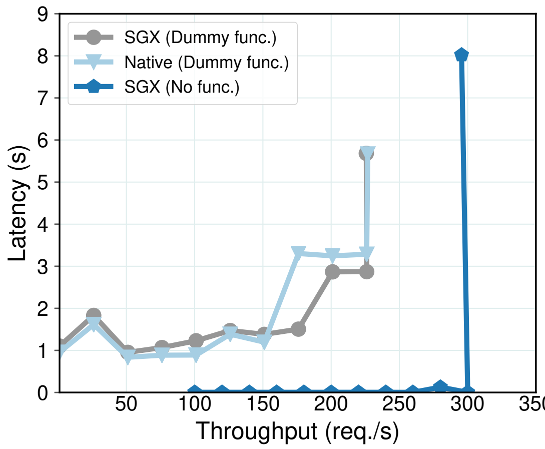

Figure 6: System latency (95th percentile) with SGX and native API Gateway with and without OpenWhisk functions.

inside serverless platforms, as these platforms are designed for very different workloads. Current research shows that OpenWhisk can add up to a second of latency; accordingly, as our evaluation of Clemmys shows, the OpenWhisk overhead completely masks the overheads of SGX. The results, however, are not representative of the workloads with very long runtimes, or if OpenWhisk system latency improves. In this case, the overheads of SGX would not be masked by the OpenWhisk overheads, as the latter do not depend on the workload. Previous research shows that SGX can cause applications to become up to ~100 times slower during EPC paging; even without resource starvation, SGX application may see ~10% SGX overhead [31] due to memory encryption and interrupts. These overheads can become even greater if side-channel or Spectre attack mitigations are deployed.

Currently, Clemmys has a number of limitations. We have already mentioned in §5 the inability to update the secure configuration after application startup. Also, OpenWhisk and Palaemon management systems are not integrated. Thus, when an operator creates a new function chain in the OpenWhisk, it is not automatically added to the Palaemon, and will not run until she manually updates the chain using the Palaemon API. The same is true for defining keys, signing user certificates, and so on. These inconsistencies can lead to denial of service when, for example, function chain configurations are separately changed in Palaemon and OpenWhisk.

OpenWhisk supports a number of action triggers, including periodic activations and activations based on, for example, Kafka streams. In our work, we have focused on the actions triggered via REST interfaces. In future, Clemmys can be extended to support these triggers as well.

The function chaining protocol so far supports only linear processing chains, without branching and loops. In case of more advanced chaining graphs [17], the format of the function chaining protocol would have to be redesigned. The format is not OpenWhisk-specific, and can be reused with other FaaS platforms.

Clemmys does not depend on SCONE conceptually: the same functionality can be be implemented with Graphene-SGX and Intel SGX SDK.

Security discussion. In this section, we argue why the security design of Clemmys is sound. First, we observe the communication in the system: at each hop between system nodes the communication is encrypted, either with TLS or with Clemmys function chaining protocol. Thus, neither eavesdropping nor message modification can cause violation of security guarantees.

Likewise, each node that handles the plaintext data from the user is protected inside Intel SGX enclave. All nodes that are running without SGX get access to the user data only in the cyphertext form. Thus, observation and modification of the data is either not possible (access denied by Intel SGX), or cannot affect the data confidentiality and integrity, as neither message plaintext nor keys are available. To guarantee integrity and confidentiality of the data from third parties (e.g. Amazon S3), it must be accessed only via TLS.

With these defences, the attacker is still able to perform two attacks: send the message to the wrong function in a chain, and send an outdated message to the same function chain (rollback attack).

We have shown in the evaluation (§6) that our protocol protects the system from the first attack: when the wrong function receives the cyphertext message, it will verify that the destination index in the chain of the message matches the identity of the current function. If this is not the case, the function will reject the message. Thus this attack should not be possible. As far as the rollback attack is concerned, it is still possible, and should be countered at the application level.

8 Related Work

Researchers and the industry have proposed several FaaS platforms. The first production system built with serverless architecture is AWS Lambda [3]; soon, several other open-source [1, 8–10] and commercial [4–6] platforms appeared. Yet, none of them protects against privileged attackers.

The work that most closely resembles ours is S-FaaS [14], a trustworthy and accountable FaaS system. Like Clemmys, S-FaaS is built on Apache OpenWhisk, and uses attastation scheme similar to that of Palaemon. S-FaaS implements a different key management scheme without special provisions for function chaining, and focuses on providing trusted resource usage accounting to the clients.

SecureStreams [21] proposes a system for secure stream computations. In this system, ZeroMQ and symmetric encryption are used for secure network communication, while the stream processing functions are implemented in the Lua programming language. In our work, we also rely on custom protocol with symmetric encryption, while the function can be implemented in several programming languages.

Opaque [36] is a Spark-based data analytics platform with data-oblivious processing functions for untrusted cloud platforms. Unlike Clemmys, it ensures that no information about workload is leaked through metadata; in Clemmys, this property is considered to be out of scope, as Clemmys deals with much more generic workloads. Pesos [22] and Speicher [16] proposed secure storage solutions based on Intel SGX, which can be potentially used for the secure storage layer.

Performance issue of serverless computing has been in the focus of research for quite some time already. Liang Wang et al. has studied and reverse engineered resource management policies of several commercial cloud platforms and discovered that they do not achieve the claimed levels of scaling [34]. Specifically, SOCK [26] and Cntr [32] have suggested the use of lean containers to reduce the function startup cost. SAND [13] achieves low startup time using fine-grained function sandboxing and high-locality message queuing and storage. These optimizations are orthogonal to those proposed in our work.

9 Conclusion

We presented Clemmys, a secure platform for serverless computing, that allows users to ensure confidentiality and integrity of functions' sources and data. Clemmys achieves these properties by building on Apache OpenWhisk, a popular serverless platform, Palaemon, a key management solution for Intel SGX, and Scone, a Intel SGX TEE framework for unmodified POSIX applications. As part of Clemmys, we propose a message encryption scheme that ensures integrity of the function chains. We have evaluated Clemmys in different scenarios and show that Clemmys achieves minimal throughput and latency overhead while providing protection against privileged adversaries.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. This project was funded by the European Union's Horizon 2020 program under grant agreement No. 690588 (SELIS), and by BMBF (03ZZ0517A, Fast-Cloud).

References

[1] Apache OpenWhisk is a serverless, open source cloud platform. https://openwhisk.apache.org. Last accessed: April, 2019.

[2] ARM TrustZone. https://developer.arm.com/technologies/trustzone. Last accessed: April, 2019.

[3] AWS Lambda – Serverless Compute. https://aws.amazon.com/lambda/. Last accessed: April, 2019.

[4] Azure Functions—Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/. Last accessed: April, 2019.

[5] Cloud Functions - Event-driven Serverless Computing. https://cloud.google.com/functions/. Last accessed: April, 2019.

[6] Cloud Functions - IBM Cloud. https://www.ibm.com/cloud/functions. Last accessed: April, 2019.

[7] HAProxy—The Reliable, High Performance TCP/HTTP Load Balancer. https://www.haproxy.org/. Last accessed: October, 2018.

[8] Kubeless — Kubernetes-native serverless framework. https://kubeless.io. Last accessed: April, 2019.

[9] OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com. Last accessed: April, 2019.

[10] Serverless Functions for Kubernetes - Fission. https://fission.io. Last accessed: April, 2019.

[11] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4):79:1–79:35, July 2018.

[12] G. Adzic and R. Chatley. Serverless computing: Economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 884–889, New York, NY, USA, 2017. ACM.

[13] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 2018. USENIX Association.

[14] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel SGX. *CoRR*, abs/1810.06080, 2018.

[15] AMD. Architecture programmers manual: Volume 2: System programming, 2018.

[16] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[17] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 89–103, New York, NY, USA, 2017. ACM.

[18] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009.

[19] I. Corporation. Attestation service for intel® software guard extensions (intel® sgx): Api documentation. https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf, accessed on 20/09/2018.

[20] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[21] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. Securestreams: A reactive middleware framework for secure data stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 124–133, New York, NY, USA, 2017. ACM.

[22] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2018.

[23] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBounds: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[24] 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*, HASP 2016, pages 10:1–10:9, New York, NY, USA, 2016. ACM.

[25] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[26] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[27] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[28] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer. Fex: A Software Systems Evaluator. In *Proceedings of the 47st International Conference on Dependable Systems & Networks (DSN)*, 2017.

[29] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 227–240, Boston, MA, 2018. USENIX Association.

[30] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[31] S. Arnautov et al. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[32] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[33] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 2:1–2:14, New York, NY, USA, 2018. ACM.

[34] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[35] L. Zhang, Y. Zheng, and R. Kantoa. A review of homomorphic encryption and its applications. In *Proceedings of the 9th EAI International Conference on Mobile Multimedia Communications*, MobiMedia '16, pages 97–106, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[36] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 2017. USENIX Association.