Trust More, Serverless

Stefan Brenner
TU Braunschweig, Germany
brenner@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig, Germany
rrkapitz@ibr.cs.tu-bs.de

ABSTRACT

The increasingly popular and novel Function-as-a-Service (FaaS) clouds allow users the deployment of single functions. Compared to Infrastructure-as-a-Service or Platform-as-a-Service, this enables providers even more aggressive and rigorous resource sharing and liberates customers from tedious maintenance tasks. However, as a crucial factor of cloud adoption, FaaS clouds need to provide security and privacy guarantees in order to allow sensitive data processing.

In this paper, we investigate securing FaaS clouds for sensitive data processing, while respecting their new features, capabilities and benefits in a technology-aware manner. We start with the proposal of a generic approach for a JavaScript-based secure FaaS platform, then get more specific and discuss the implementation of two distinct approaches based on (a) a lightweight and (b) a high performance JavaScript engine. Our prototype implementation shows promising performance while efficiently utilising resources, thereby keeping the penalties of the added security low.

CCS CONCEPTS

• **Security and privacy → Systems security**; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

Trusted Function-as-a-Service, Intel SGX, Serverless Cloud

ACM Reference Format:
Stefan Brenner and Rüdiger Kapitza. 2019. Trust More, Serverless. 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/10.1145/3319647.3325825

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
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/10.1145/3319647.3325825

1 INTRODUCTION

The increasing popularity of Function-as-a-Service (FaaS) is due to its benefits as this approach allows cloud customers the deployment of small standalone functions—often called *Lambdas*—in a flexible way. It is the natural succession of cloud paradigms, shifting from Infrastructure-as-a-Service (IaaS) towards Platform-as-a-Service (PaaS) clouds, to reduce the maintenance tasks of a cloud customer and offload more maintenance tasks to the cloud provider [7, 14]. This paradigm allows customers to liberate themselves even more than with previous approaches from the burden of investing in and maintaining a compute platform, and delivers flexible and fine-grained accounting, as only actually executed Lambda invocations appear on their bill. Furthermore, this paradigm simplifies the development, as scalability is included automatically—the cloud provider will make sure enough instances of a single function are created on different machines to cope with the current demand.

Just like for IaaS and PaaS clouds, security is a crucial factor that affects cloud adoption in case of FaaS clouds as well [15, 18]. Only by providing a Trusted Execution Environment (TEE) inside the cloud—an execution environment that protects code and data from unauthorised access—the cloud customer is relieved from fully trusting the cloud provider. This is of special interest when the cloud provider is a globally operating company that might not be bound geographically or legally to the customers environment. In general, sensitive data processing as in medical applications with patient data or governmental or police authority's IT services, is only possible by guaranteeing confidentiality and integrity in the cloud. In addition, a TEE in the cloud allows the removal of huge amounts of code from the Trusted Computing Base (TCB) of a cloud-based application, such as the cloud provider's software stack for managing the infrastructure and controlling the accounting.

We believe that the demand for security and the nature of FaaS clouds—the deployment of single standalone functions in the cloud—are a natural fit for trusted execution technology such as Intel Software Guard Extensions (SGX). For example, an Intel SGX secure enclave was intended to be akin to a library that comprises a few trusted functions being called from the untrusted main application [21].

However, naively porting existing FaaS platforms to SGX is not a valid solution as this would lead to major performance problems and an inefficient design that does not scale

© ACM, 2019. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published on the 12th ACM International Systems and Storage Conference (SysTor '19). https://doi.org/10.1145/3319647.3325825

for large numbers of Lambdas. For example, most existing FaaS architectures work with interpreted languages (such as JavaScript, Python or Java) and spawn each Lambda and the required runtime system inside their own Docker container [14]. This duplicates the runtimes in memory, even for Lambdas written in the same interpreted language and multiple instances of the same Lambda. While this approach is already an inefficient memory usage without trusted execution involved and only accepted for its strong isolation guarantees between the Lambdas, due to the limited trusted memory of Intel SGX this also leads to major performance problems in such a trusted environment [6, 12].

In this work we target a secure and trusted FaaS platform that maintains the original benefits of FaaS platforms and exploits the trusted execution technology in an efficient way tailored to its specific characteristics. Thereby, the challenges to reach a secure FaaS platform that must be retained comprise the following: Function-level isolation is required to isolate functions from each other; especially from mutually distrusting customers. In addition, functions should be provisioned on-demand and quickly, i.e. be able to perform a fast cold start, with low user-facing latency especially for rarely used Lambdas. Also, we need to be able to use available resources efficiently especially to support scalability of the system to large numbers of customers that deploy Lambdas, and many users simultaneously issuing requests. Finally, we want customers to be able to establish trust in the platform and allow sensitive data processing without trusting the cloud provider.

Implementing Lambdas using a language that compiles directly to native code would be desirable, as in that case no large runtime like an interpreter is required and Lambdas can be assigned to their own secure enclave, which leads to a rather strong isolation. However, this prevents any sharing of code because shared memory between enclaves is not foreseen. Additionally, deploying multiple Lambdas into the same enclave in order to allow sharing (e.g., sharing the libc) is not possible, as native code would need to be isolated in that case using a large and complicated sandbox mechanism [16]. Also, any libraries used by native Lambdas would need to be ported to run inside the enclave as well.

In contrast to native code, code written in interpreted languages is usually sandboxed by the respective runtime environment. However, even then there are library dependencies and most libraries rely on native components as for example it is the case for Lua or Python. Due to its originating as a self-contained language running small scripts in a Browser, JavaScript is one interpreted language that has a lot libraries written in pure JavaScript code without any native dependencies. With that assumption, we can provide a platform that runs multiple *pure* JavaScript-based Lambdas on top of a single JavaScript engine inside the same enclave, and therefore shares the relatively large interpreter between multiple Lambdas even from different Lambda providers. Google's V8 JavaScript engine for example is written from the ground up with the idea of mutually distrusting JavaScript components in mind—isolated from each other by so called *v8 Isolates*[2].

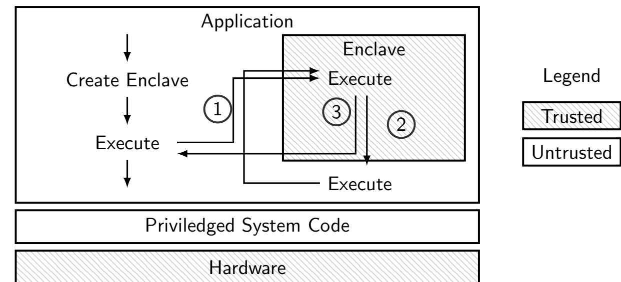

Figure 1: SGX enclave interaction.

The contributions of this paper comprise the following:

- Investigation towards the design and implementation of a secure FaaS platform with trusted execution based on Intel SGX that retains the benefits of FaaS and respects the characteristics of trusted execution as offered Intel SGX at the same time.
- Proposal of a generic architecture for a secure FaaS platform supporting JavaScript Lambdas.
- Two implementation variants of this generic FaaS architecture, (a) a lean platform based on the lightweight *Duktape* JavaScript engine, and (b) a high performance platform based on Google's V8 JavaScript engine.
- The evaluation of the performance of those two platforms, especially the latency overhead for trusted Lambda requests compared to an untrusted platform and the trusted platform's overall throughput.

Our paper is structured as follows: In Sec. 2 we describe the fundamentals, what Lambdas are and their characteristics, followed by the description of our design decisions towards a secure serverless cloud platform in Sec. 3 and a brief description of several implementation details in Sec. 4. Finally, we evaluate our two platforms in Sec. 5, discuss related work in Sec. 6 and conclude the paper in Sec. 7.

2 BACKGROUND

2.1 Intel Software Guard Extensions

Intel SGX [21] is a processor instruction set extension that allows the creation of an x86-based Trusted Execution Environment (TEE)—a so called *secure enclave*. Confidentiality of enclave memory is protected by transparent encryption done within the CPU, i.e., plain text is only available inside the CPU package. Valid interaction with an enclave is only possible via explicit enclave calls (ecalls) to enter an enclave

and outside calls (ocalls) to call out of the enclave. Fig. 1 depicts the interaction of an untrusted application part with an enclave, comprising an ecall ①, an ocall ②, and finally, an ecall-return ③ to the untrusted application.

In general, enclaves are bound to a user process—even multiple enclaves in one user process are possible. However, if all enclaves on a platform do not fit into the so called Enclave Page Cache (EPC), a special form of reserved memory of at most 128 MB that backs all enclave pages on a platform, swapping to regular RAM is required. This process induces high performance overhead, as it requires re-encryption of the memory page's contents, as well as measures to ensure integrity and prevent replay and rollback attacks.

As enclaves can only run in user space, SGX-based applications still depend on the underlying Operating System (OS) to cooperate. This includes the enclave's memory management by the untrusted OS. Hence, SGX naturally can not prevent denial of service attacks by privileged code.

Programming Support with Intel's SDK Intel offers an SDK to handle the enclave's life cycle and memory management. It also comprises the Enclave Description Language (EDL) that allows the specification of the ecalls and ocalls the enclave supports and the "Edger8r" tool which generates code for ecall and ocall stubs. Finally, enclaves are compiled and linked together with the generated code to a shared object, which is loadable as an enclave by using SDK functions.

SGX Remote Attestation Intel supports remote attestation of SGX enclaves without requiring physical access to the hardware, in order to verify genuine SGX-capable CPUs that run secure enclaves [5]. The procedure requires the enclave to generate a report of its properties such as the creation process of the enclave and a hash of the code running inside it (`MRENCLAVE`), that is signed by a Intel-provided service enclave running on the same platform. This is called *Local Attestation* and results in a *Quote* that is forwarded to the attester, who is then able to verify the Quote at the Intel Attestation Service (IAS) provided by Intel. Inside the messages exchanged between the enclave and the attester, a small amount of arbitrary payload data can be enclosed as well, that is also covered by the security mechanisms of the attestation procedure ensuring authenticity and integrity of the data. This can be used for example to exchange a key between the enclave and the attester.

2.2 An SGX-aware Threat Model

We assume a typical threat model for SGX enclaves [8], specifically in an untrusted cloud environment: an attacker has full—even physical—control over the server hardware and software environment. This includes control over the OS and all code invoked prior to the transfer of control into the SGX enclave. The attacker's goal is to break confidentiality or integrity of the code running in the SGX enclave.

Availability threats such as crashing an enclave are not of interest, as, inherently to the SGX paradigm, the hosting OS can stop enclave execution arbitrarily and at any time. Also, we assume there are no replay attacks on sealed data of our enclave, or withholding of sealed data, as these problems belong to their own research field of state-continuity [10].

We do not consider side-channel attacks [13, 19, 29] and assume that enclaves do not possess any security-relevant vulnerabilities that may lead to data leakage or integrity breaches. Also, we assume that basic building blocks running inside secure enclaves such as the Intel SGX SDK are bug-free, and trust the design and correct implementation of the CPU package and the SGX instructions including all cryptographic operations done by SGX. While the recent Meltdown [20], Spectre [17] and Foreshadow [25] bugs have shown that hardware is not impeccable, we assume such bugs are fixed by microcode updates.

2.3 The Lambda Model—What are Lambdas

FaaS can be seen as the successor of earlier cloud computing approaches like IaaS and PaaS. While those provide virtual machines or containers to their users, the FaaS paradigm handles single standalone functions. In this paper we call those *Lambdas*, inspired by the first FaaS platform by Amazon.

Initially, the idea of IaaS was to increase resource utilisation and provide ease of scalability to customers, while at the same time taking the risk of high investments off their shoulders. This idea evolved in PaaS that further shifted the management overhead off the cloud users to the cloud provider and allowed more efficient resource sharing. FaaS builds upon that thought and allows the customers to completely stop thinking about single server machines and their management—thus, it is also called the *serverless* paradigm.

Lambdas are small inherently stateless functions, usually written in interpreted languages like JavaScript or Python [14]. They are provided by the FaaS user or customer, and executed in isolated environments like Docker containers on the cloud provider's machines [27].

Large Lambda applications are comprised of multiple Lambdas forming a holistic application by interconnecting the Lambdas that can use a common database or other forms of persistent storage. Lambdas can be called directly or triggered due to events, such as storing a file in a data store. Essential to Lambdas is their ability to be started quickly and also to automatically scale across multiple machines according to the current demand.

3 TOWARDS SECURE FAAS

In this section we describe our approach of a secure FaaS platform using Intel SGX. In particular we discuss how the

trusted computing resources can be used efficiently, while preserving the characteristics and advantages of FaaS over other cloud computing paradigms like IaaS and PaaS.

We first analyse the requirements of such a platform, regarding the user-facing features and the provider-facing capabilities. Then, discuss crucial aspects like scalability, cold start latency and resource isolation and efficiency. Next, we investigate the suitability of various interpreted languages for usage in such a platform, before we describe our proposed platform's architecture for the lightweight Duktape and the fast Google V8 JavaScript engine. Finally, we outline how attestation of Lambdas can be done by users of our platform.

3.1 Secure FaaS Requirements

In general, a Lambda platform should be able to run larger numbers of Lambda scripts in parallel and isolated from each other, also supporting auto-scale depending on the current load. Lambdas should quickly be spawned if not already running, in order to achieve low response times for Lambda requests. Also, available system resources should be used efficiently, therefore a low memory footprint of each single Lambda and its context is favourable, in order to be able to hold many Lambdas in memory before being forced to evict them. In addition, execution of many Lambdas in parallel should be supported not only to fully use multi-core CPUs but also smoothen IO delays.

For a Lambda platform to become trustworthy, a few more aspects need to be considered. Firstly, the confidentiality and integrity of network communication must be ensured, otherwise there would be no point of using trusted execution on the server-side. Secondly, as the cloud provider is not trusted and the data that is processed is considered sensitive, the Lambda's execution state and all processed data must be protected. Usage of trusted execution technology like Intel SGX naturally leads to this, as the enclave memory is encrypted, and thus, not available to the cloud provider. But even in case of using other (weaker) trusted execution technology like ARM TrustZone, the data processed by a Lambda would not be (easily) accessible by the cloud provider, except for physical attacks such as cold-boot attacks.

In order to achieve good performance with a trusted Lambda platform as outlined above, a few new factors need to be incorporated which are specific to the trusted execution technology being used—Intel SGX in this case. In general, the enclave size should be as small as possible in order to prevent SGX paging as long as possible, as this induces a high performance impact [6, 12]. Also, Brenner et al. [11] have shown, that using lesser number of enclaves with the same code and responsibility improves the performance, not only because libraries can be shared but also due to a more efficient CPU cache usage. Furthermore, as Lambdas are usually written in interpreted languages and comprise of relatively little code compared to the required runtime, sharing the runtime between multiple Lambdas is quite beneficial regarding memory usage. This leads to an architecture that executes multiple (competing) Lambdas inside the same runtime in the same enclave, and thus, additional strong isolation between Lambdas is required in order to ensure they can not access each other's data and monopolise resources. This includes isolation of subsequent requests to the same Lambda as well.

3.2 Suitability of Interpreted Languages

In the previous section we described generic requirements of any FaaS platform and additional aspects specific to a trusted FaaS platform. This section discusses the individual suitability of various programming languages and their runtimes if applicable as the basis of our secure FaaS platform.

In principal, Lambdas implemented in compiled languages (e.g., C/C++ and Rust) would be advantageous, as their resource footprint is relatively small compared to interpreted languages that do require a full interpreter in order to execute. However, Lambdas are traditionally written in interpreted languages and we would like to support execution of existing Lambdas with minimal or at least automatable effort on top of our platform. In addition, native Lambdas would have to be ported and recompiled to be executable inside a secure enclave, as no system calls are available there without further measures. Even though we assume that Lambdas do not establish their own socket connections or access the file system directly, even simple actions like requesting the current time (`gettimeofday()`) may require system calls. This poses a problem also for library dependencies of Lambda code, as those libraries would have to be ported as well. Only approaches like Haven [8], Graphene [24] and SCONE [6] do allow execution of unchanged native applications inside secure enclaves, at the cost of a large runtime for example comprising a library OS inside the enclave. Also, isolation of native code is hard, as arbitrary memory locations can be accessed if no complex sandboxing is implemented inside the secure enclave as well.

Code components written in interpreted languages could run without further changes inside a secure enclave, as long as the interpreter is available there. In addition to that, if the interpreted code has no library dependencies with native code parts, pure interpreted execution can be isolated from code in the same address space relatively easily, as memory access is controlled by the runtime. However, interpreted code is usually much slower than native code, especially if not accelerated by Just-In-Time (JIT) compilation.

Due to the above advantages of interpreted languages, their popular usage in FaaS systems, and the security issues of native code, we investigated several interpreted language environments and their suitability for usage in a secure FaaS

scenario. Even though most runtimes for interpreted languages offer at least some notion of a "context" to isolate code and share the runtime with others, they heavily rely on libraries and most of those libraries contain large fractions of native code components. This applies for example to Lua and Python. However, JavaScript has been originating from web browsers initially, that by design required the code to be platform independent and more or less self-contained. Therefore, many available JavaScript libraries are written in pure JavaScript code without any native dependencies. In addition, many serverless applications are written in JavaScript as JavaScript is a trending and popular language and all existing FaaS platforms (e.g. OpenWhisk[1]) support at least JavaScript amongst other languages [14]. For those reasons, we aim at supporting JavaScript-based Lambdas in the architecture that we describe in this paper. However, we limit our platform to the execution of pure JavaScript code without any native components, due to the negative side effects of native code in shared enclaves as described above.

Nevertheless, even for JavaScript as a promising Lambda language candidate, there are still multiple options to execute JavaScript code. MuJS[2] was the first interpreter we looked into. While being extremely small and resource-efficient, MuJS has only quite limited ECMAScript 2015 support thereby inhibiting the use of modern JavaScript programming idioms. In contrast, the Duktape[3] JavaScript engine provides good—but still partial—support of ECMAScript 2015 and other features such as the ES2015 TypedArray and Node.js Buffer bindings, for example. Supporting ECMAScript 2015 would be beneficial, as regular JavaScript code can be automatically transpiled to ECMAScript 2015, but not necessarily to older versions as well. Finally, there is Google V8 as one of the most modern JavaScript engines with features for high performance such as JIT compilation and more sophisticated garbage collection approaches. While providing the most holistic language support, Google V8 with $\approx$ 1.3 million Source Line of Code (SLOC) is by far also the largest of the engines we investigated in terms of its TCB and memory footprint. However, according to our measurements (*3dcube* and *base64* benchmark of the JetStream suite[4]) Google V8 performs about 30× to 84× better than Duktape.

3.3 Lambda Library Dependency Bundling

It is unrealistic to assume Lambdas will get along without any dependencies. Hence, there must be a way to support library dependencies of Lambdas running on our platform.

Instead of loading libraries on demand from the outside, dependencies could also be bundled with the Lambda code into a standalone script. The advantage of this is, that the Lambda is represented by a single file with a signature inside the Lambda storage system, and can easily and quickly be loaded into the enclave with a single call. Also, in that case, the signature naturally includes the libraries used by the Lambda. In order to achieve this, we used *webpack*[5]. Webpack allows automatic resolving of calls from JavaScript code to the `require()` function, and downloads and bundles all required library dependencies recursively with the Lambda into a single standalone file.

[1]https://openwhisk.apache.org
[2]MuJS JavaScript Engine http://mujs.com/
[3]Duktape JavaScript Engine https://duktape.org/
[4]JetStream JavaScript Benchmark Suite: http://browserbench.org/JetStream/

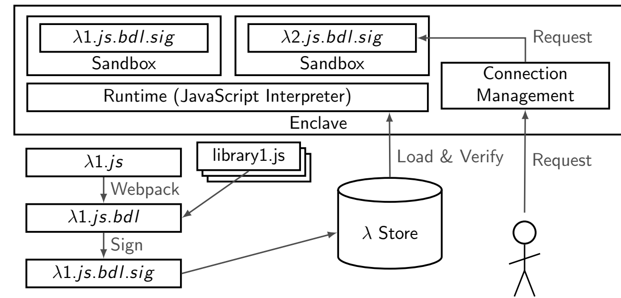

Figure 2: Secure FaaS Platform Generic Architecture.

3.4 Generic Secure FaaS Architecture

The generic architecture of our secure FaaS platform shown in Fig. 2 is in principle independent from the JavaScript interpreter being used. Still, the interpreter is the heart of the platform, running once inside the enclave, and is shared between all Lambdas executed within that enclave. All Lambdas are executed in their own context, in order to be able to execute Lambdas in parallel and provide reasonable isolation between them, even if the same Lambda script is executed multiple times in different contexts in order to improve performance by parallel execution in high load situations.

In general, Lambdas are stored outside the enclave in the form of a signed (or even encrypted) bundle of the Lambda's JavaScript code. This bundle is being loaded on demand by the platform from the untrusted storage into a newly created context and prepared for being called by users. On load, the Lambda bundle's signature (The Lambda bundling process is described in Sec. 3.3) is being verified by the platform in order to ensure that only Lambdas correctly signed by valid customers are executed on the platform.

After a Lambda is loaded and verified a new context is created for its execution with the JavaScript interpreter. This ensures the Lambda is executed independently and isolated from other Lambdas and can run in parallel with other Lambdas. On high load on a single Lambda it may also be beneficial to instantiate multiple contexts for the same Lambda, in which case the Lambda is only loaded once from the outside, but multiple independent contexts are created from it.

[5]webpack.js https://webpack.js.org/

A Lambda is loaded only on-demand, when a request for this Lambda arrives from a user and the Lambda is not yet present in the enclave. A new connection also requires a connection context to be created that stores connection-specific data such as the TLS session keys amongst others. Such a connection context is not bound to a specific Lambda and a Lambda context is even independent from a user connection.

3.5 Adjustment to Current System Load

In order to adjust the number of created Lambda contexts to the current load situation on the system, we introduce a μ-value, that describes the pressure on a Lambda context. The μ-value is calculated by the number of requests for each individual Lambda script in a given time frame, divided by the amount of "waits" that are required to find an available Lambda context. When a request for a specific Lambda script is being processed, the platform will first try to find an unused context for this script and wait until a context is being released by another thread if none is available. If too many "waits" are required, the μ-value will decrease, and the platform can spawn new contexts for a Lambda script once the μ-value falls below a configurable threshold. This ensures the optimal amount of contexts is created for a Lambda script, which varies depending on how much computation the Lambda does and the number of requests per second.

The above μ-value adjusts the concurrent contexts of one Lambda on a single host. Coarser-grained adjustment on a larger scale is additionally supposed to be done by the untrusted cloud platform, balancing load across different machines and enabling scalability of Lambda applications. This can be done by the untrusted cloud provider using traditional technical means, and respecting properties like locality in established placement policies.

3.6 Using the Duktape JavaScript Engine

This section details the implementation of our generic architecture with the Duktape JavaScript engine as the interpreter of the Lambdas—this system is called *Secure DukTape Lambda Platform (SecureDuk)* throughout this paper. In this case, the Duktape engine is embedded into an Intel SGX SDK enclave as a library, and is being used by our platform application to interpret the Lambdas that are loaded from the outside Lambda store. This instance of our generic architecture uses Duktape JavaScript contexts for Lambda isolation.

Requests issued by users are transmitted via a TCP socket opened by our application and are integrity- and confidentiality-protected during their transmission by TLS—the TLS endpoint resides *inside* the enclave.

The so called *LambdaManager* component inside the enclave is responsible for managing the life cycle of our Lambda contexts. When a context is required for a specific Lambda script, the script is only loaded from the outside Lambda store if not yet available inside the enclave. Contexts are also created on demand and reused for multiple invocations of the same Lambda. In case of very high load, even multiple contexts for the same Lambda can be instantiated (See Sec. 3.5 for more details on the μ-value).

3.7 Using the Google V8 JavaScript Engine

This section describes the implementation details of our generic architecture with the Google V8 JavaScript engine—which we call *Secure Google V8 Lambda Platform (SecureV8)* in this paper. Its architecture is a specific case of our generic architecture with the Google V8 JavaScript engine running on top of SGX-LKL, both inside the enclave. To achieve this we used the SGX-LKL[6] project that combines the Intel SGX technology and Linux Kernel Library (LKL), which is the Linux kernel as linkable library. SGX-LKL resembles other approaches for execution of legacy applications inside enclaves (e.g., by using a library-OS) such as [6, 8, 23, 24]. The enclave is not created by the Intel SGX SDK but a custom re-implementation of the SGX enclave creation and management process. With SGX-LKL there is user-level threading inside the enclave, as well as support for synchronisation and coordination of multi-threaded applications by using mutexes and conditional variables solely inside the enclave. In addition, SGX-LKL allows its guest application to issue system calls, that are processed asynchronously by threads outside the enclave. We built our *SecureV8* platform on top of SGX-LKL and linked it against the Google V8 JavaScript engine compiled for the *musl libc*[7] library, as this is required to run the application inside an SGX-LKL-based enclave.

Similarly to *SecureDuk*, also *SecureV8* opens a TCP connection in order to listen to user requests. However, in this case the system calls related to the socket are handled by the asynchronous system call queuing mechanism of SGX-LKL and issued to the host kernel running outside the enclave. Still, confidentiality and integrity is protected by a TLS encryption that terminates inside the enclave.

The *SecureV8* application also maintains a *LambdaManager* component, just like *SecureDuk*. However, Lambdas are not only isolated by V8 contexts, but by V8 isolates that each comprise exactly one context in our case. This leads to a stronger isolation of Lambdas in contrast to *SecureDuk*, as V8 isolates have originally been designed for mutually distrusting scripts running in different web browser tabs.

3.8 Key Management & Bootstrapping

In Fig. 3 we describe key management and trust relationships of our platform. The figure introduces the following *entities*: the *cloud provider* owns the hardware and runs the enclave and the platform software (either *SecureDuk* or *SecureV8*).

[6]SGX-LKL at Github: https://github.com/lsds/sgx-lkl
[7]musl libc https://www.musl-libc.org/

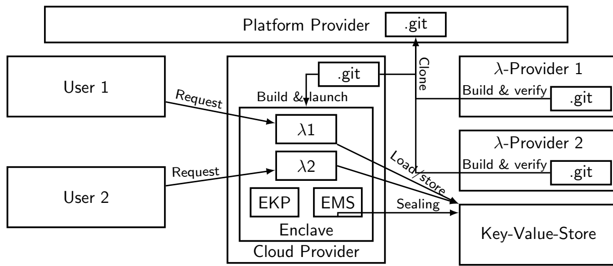

Figure 3: Key Management.

The *Lambda providers* run their own Lambdas on top of the platform. The *platform provider* is the entity that develops and distributes the secure Lambda platform's software. Finally, the *users* issue requests and use the Lambdas via secure TLS connections terminating inside the enclave. Not all entities must strictly be distinct, but some of them may be the same, for example, a Lambda provider can also be a Lambda user at the same time (i.e., in case of Lambda chaining).

Furthermore, we define the following two keys: the Enclave Master Secret (EMS) is a symmetric key used for confidentiality protection of data stored in the untrusted key-value store (KVS) hosted by the cloud provider. This EMS key is generated by the enclave and can be migrated from one enclave instance to another after successful mutual attestation, allowing the cloud provider to scale the platform to multiple instances. The EMS is stored itself inside the KVS (after sealing) to allow correct enclaves to automatically bootstrap the system. In addition, an enclave will generate (at enclave start) the Enclave Key Pair (EKP) used for securing the connection between Lambda providers and the enclave.

The cloud provider initially acquires the source code of the platform software from the platform provider and verifies it to ensure it will not harm her infrastructure. After a successful verification, the cloud provider builds and deploys the platform software and maintains its availability.

In order to establish trust into the Lambda platform, the Lambda provider acquires the platform's source code and verifies and builds it in order to generate the expected hash value (`MRENCLAVE`) of the platform's binary. Then, the Lambda provider attests the platform running in the cloud by remote attestation with a nonce (for freshness) and the known hash value of the binary. The platform returns an SGX attestation quote (c.f. Sec. 2), comprising the public key of the EKP and a signature that can be verified using the IAS containing a hash over the above mentioned nonce from the attester and the EKP's public key. Afterwards, the Lambda provider can send a key encrypted with the EKP public key to the platform. This establishes a secure connection between the enclave and the Lambda provider, which allows uploading Lambdas and TLS keys stored by the platform inside the untrusted KVS encrypted with the EMS.

After a Lambda is uploaded and stored inside the KVS along with at least one TLS key per Lambda provider, a Lambda is ready to process user requests. Requests are addressed to a specific subdomain identifying the Lambda provider, which allows the user to implicitly detect that the Lambda provider has successfully attested the platform, as only then a valid TLS connection is possible. In addition, this approach even supports multiple different Lambda providers in the same enclave, as each Lambda provider uses her own TLS key. By using Server Name Indication (SNI) the platform can maintain multiple TLS endpoints for distinct subdomains under the same socket, all terminating inside the enclave.

The above approach allows the execution of Lambdas that are implicitly attested by the users issuing requests. Integrity of the Lambda code can be protected by an Keyed-Hash Message Authentication Code (HMAC) stored with the Lambda code inside the KVS. Note that the Lambda is already integrity-protected during transmission by TLS and inside the enclave by the Intel SGX memory encryption.

In our concept the cloud provider is not to be trusted by any other entity. We allow multiple Lambda providers on the same platform with distinct keys, and enable users to only trust selected Lambda providers. Derived from the fundamental trust into the platform, we support Lambda-specific sealing of data (into the KVS) using a Lambda-specific key derived from the EMS and a hash value of the Lambda's code.

3.9 Security Considerations

Lambdas must be detained from reaching outside their projected environment and harm the cloud provider or the platform. Furthermore, it is crucial to ensure that one Lambda can not access any data of another Lambda. For this reason, Lambdas are isolated from each other using container-based isolation mechanisms in many existing FaaS platforms—AWS and OpenLambda use Docker containers for example [14, 26].

In addition, besides the high porting effort of native code to enclaves, one of the strongest arguments against native code even in the form of library dependencies of Lambdas is the required isolation of Lambdas. As native code components work with pointers, a large and complex sandbox or hardware mechanism must be brought in place to isolate them from each other [16]. By abandoning support for native components, isolation becomes much easier as almost all existing runtimes for interpreted languages already posses a notion of (isolated) contexts to run multiple scripts independently from each other. Even though previous executions of Lambdas will leave pre-owned objects behind, there is no way to access them from interpreted code before the garbage collector eradicates them due to the lack of references.

The problem with existing FaaS platforms in an SGX scenario is that the interpreter can not be shared between multiple Lambdas and must be instantiated for each Lambda. This leads to high memory consumption which is particularly difficult as SGX can only maintain good performance of the transparent memory encryption for small memory ranges (≈128 MB as mentioned in Sec. 2). The only option with SGX to achieve good performance is to co-locate multiple Lambdas inside one enclave using the same interpreter, however, isolation between Lambdas is essential in this case. For isolation, process-based isolation would be the best option, as this is considered the strongest mitigation against a Speculative Side-Channel Attack (SSCA) [3], but this prevents sharing of the interpreter between Lambdas.

Our vision is to resolve this dilemma with policies negotiated by cloud customers and the provider that specify the required security-levels of Lambdas. For highly sensitive Lambdas the provider could be advised to start dedicated enclaves while other less sensitive Lambdas may be co-located with (a) Lambdas of the same provider or (b) Lambdas of other Lambda providers. Obviously, highly sensitive Lambdas will lead to higher cost in that case as they require more resources, so the cloud provider can assign a price tag to the required level of security. Enforcing those policies relies on an attested and trusted base platform inside the enclave which is independent from other Lambdas running in the same enclave, as described in Sec. 3.8.

In addition to the above policies of Lambda security requirements, there are newly introduced mechanisms of the V8 JavaScript engine especially for isolation of untrusted JavaScript code and SSCA mitigation available since Google V8 v6.4.388.18 (we used Google V8 v6.7.77.0), but they add a performance degradation of up to 15%[3]. Furthermore, we make excessive use of v8 Isolates, and never execute two Lambdas inside the same v8 Isolate. This is important to benefit from the isolation capabilities of v8 Isolates and also accepted by commercial providers, e.g. Cloudflare abandoned Node.js due to its lack of v8 Isolate support and implemented a custom application with an embedded Google V8 [1, 2].

Due to the transparently encrypted and integrity-protected enclave memory by using Intel SGX, even the cloud provider, and privileged software such as the OS can not access or manipulate the secure enclave's contents. This leads to an inherent protection of the Lambda code and data inside the enclave at runtime. In addition to that, Lambdas can store data persistently outside the enclave in a secure way by applying sealing (see Sec. 3.8). Finally, communication of Lambdas with the outside world is protected by TLS, which completes the chain of protection mechanisms for Lambdas and their sensitive assets.

Another relevant security property is performance isolation of Lambdas that ensures Lambdas can not stall execution and jeopardise liveness of the platform. This can be guaranteed by a small patch to the JavaScript interpreter, that calls `yield()` regularly. In case of *SecureV8* this triggers the internal scheduler of the user-level threading of SGX-LKL inside the enclave and allows a renegotiation of resource assignment. In case of *SecureDuk* each request is handled by a distinct connection thread, therefore, liveness can be guaranteed by the untrusted cloud platform with established and mature procedures. Since a denial of service by the cloud provider can not be prevented, this poses no additional risk.

4 IMPLEMENTATION DETAILS

The implementation of our Intel SGX SDK-based *SecureDuk* platform partly utilises the *node-secureworker*[8] project, that allows offloading small parts of JavaScript applications running on Google V8 to a Duktape JavaScript interpreter running inside an SGX v1 enclave.

TLS encryption is implemented using mbedTLS library for both, *SecureDuk* and *SecureV8*. In case of *SecureDuk*, we used the mbedTLS-SGX[9] port that allows usage of the library for Intel SGX SDK-based applications. In case of *SecureV8*, we linked the mbedTLS library to our platform application that is running on top of SGX-LKL. Messages from clients are expected to be secured by TLS and contain JSON data.

Besides the required calls of the mbedTLS-SGX library, the interface of the Intel SGX SDK-based *SecureDuk* platform to the untrusted world comprises ecalls to initialise the enclave, call a Lambda function, load a Lambda's JavaScript code from untrusted storage and thread management functions.

SecureV8 is based on the Google V8 JavaScript engine which amongst other things uses a JIT compiler to increase performance. In order for JIT to work, respective memory ranges must first be writeable to store the generated code as well as later be readable and executable in order to be able to execute that code. Since SGX v1 does not allow changing permissions of enclave pages after enclave initialisation, SGX-LKL marks all pages `rwx` at the moment to be able to support JIT. As soon as SGX v2 becomes widespread available, this behaviour can be changed to only grant the minimal set of required permissions, i.e. only readable and writeable during code generation and readable and executable for execution.

5 EVALUATION

In this section we show the results of our evaluation of the two secure Lambda platforms, *SecureDuk* and *SecureV8*, based on the Duktape and Google V8 JavaScript engine. The evaluation comprises the TCB of our two platforms, their throughput and response times, as well as the working set memory footprint of the enclaves.

[8] node-secureworker https://github.com/luckychain/node-secureworker
[9] mbedTLS-SGX https://github.com/bl4ck5un/mbedtls-SGX

Table 1: Size of Code Base in Lines of Code.

	Duktape	V8
Interpreter	185,392	1,308,702
Environment	214,156	17,193,624
Platform	1,529	1,002
Sum	401,077	18,503,328

Subject to our evaluation are several different Lambdas: The *echo* Lambda only returns its input back to the platform, while the *jpeg* Lambda decodes a JPEG image provided as base64-encoded JSON as its input and returns the image as decoded bitmap as its output and the *fibonacci* Lambda calculates the 1250th Fibonacci number. In addition, we also extracted scripts from the official JetStream benchmark suite[10] and ran them as Lambdas on top of our platform (*base64* and *3dcube*). All Lambdas are bundled prior to their deployment with all the required libraries using *webpack* into one standalone JavaScript file as described in Sec. 3.3.

5.1 Trusted Code Base

In Tab. 1 we show the number of lines of code of our two platforms. In this table, *Interpreter* represents the code of the JavaScript engine itself that is being used to interpret the Lambda's JavaScript code. *Environment* stands for the required SGX SDK libraries in case of *SecureDuk*, and SGX-LKL in case of *SecureV8* inside the enclave in order to run the application. The *Platform* line shows the amount of code of our platform application (*SecureDuk* and *SecureV8* respectively) which bridges the gap between the environment and the engine and takes care of all management tasks like managing JavaScript contexts, loading Lambdas from the untrusted storage and managing their life cycle, as well as the client connection management.

As can be seen from the table, *SecureDuk* is much smaller in terms of the required source code when compared to the *SecureV8* platform. This is mainly due to the much smaller (and slower) Duktape interpreter used in *SecureDuk*, but also due to SGX-LKL required to run Google V8 in *SecureV8* which is much larger than the SGX SDK. In total the source code of the Duktape-based platform is approximately 46× smaller than the Google V8-based platform.

5.2 Working Set Memory Footprint

In order to measure the working set memory footprint of our *SecureV8* enclave, we used *sgx-perf*[11] [28]: the tool expects the enclave start address and size, and then removes all page permissions from that range and registers a custom page fault handler. Once the enclave accesses a page, the custom page fault handler is notified and resets the page permission for that page. We exclude the enclave pages touched during a warm-up phase of 60 seconds and collect the measured working set footprint for various fixed numbers of contexts. Fig. 4 shows the memory footprint of our Lambdas and proves the low memory footprint of the platform being far below the SGX paging threshold of up to 128 MB, except for the `base64` Lambda which leads to SGX paging.

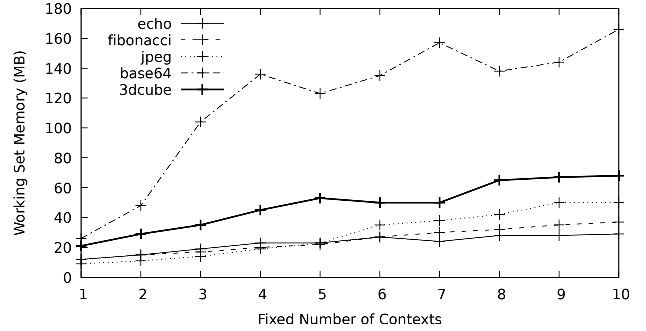

Figure 4: *SecureV8* working set footprint of Lambdas.

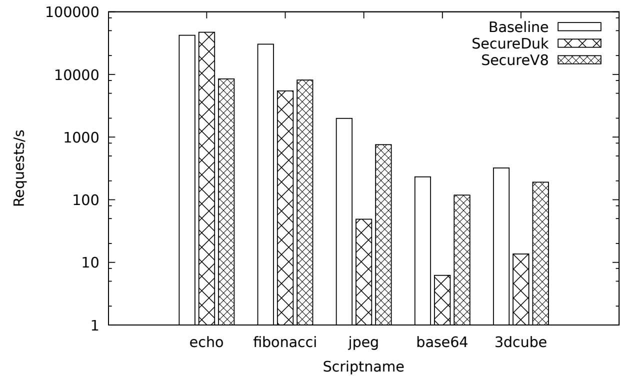

Figure 5: Throughput of *SecureDuk* and *SecureV8*.

5.3 Lambda Request Throughput

This section shows the comparison of the throughput of our two platforms against the baseline. In this case, the baseline is the Google V8-based application running on bare hardware (in an Alpine Linux Docker container in order to provide the application with the musl-libc library), and it is compared against the *SecureDuk* and *SecureV8* platform. In contrast to the earlier presented benchmarks with fixed number of contexts in Sec. 5.2, here the platform decides how many contexts are created using the μ-value as described in Sec. 3.5.

[10]JetStream JavaScript Benchmark Suite: http://browserbench.org/JetStream/

[11]sgx-perf at Github https://github.com/ibr-ds/sgx-perf

The results of this benchmark are illustrated in Fig. 5 created using the *h2load*[12] HTTP benchmark application with 12 parallel client threads and a separate warm-up phase to stabilise the results. As can be seen, the *SecureV8* platform is much faster than *SecureDuk* which achieves approximately 6% of the performance of *SecureV8* for the complex Lambdas (`jpeg`, `base64`, `3dcube`) and 67.2% for the `fibonacci` Lambda. However, in case of the *echo* Lambda, the *SecureDuk* application is even 5× faster than the baseline, emphasising the benefits of the lean *SecureDuk* platform.

5.4 Response Times

We also evaluated the requests latency of our two platforms in various cases and show the results in Tab. 2. All measurements compare the same baseline as before against *SecureV8* and *SecureDuk*. We compare the *warm* latency with the Lambda script already loaded into the enclave and a context already created for it, with the *cold* latency where the Lambda script has to be loaded from untrusted storage and a context must be first created for it. In the last row of the table, we also show the plain *overhead* of the platform including TLS encryption, enclave entering and exiting and context lookup, but *except* for the actual execution of the Lambda's JavaScript code. In addition, all measurements are done a) for a new connection, thereby including the TLS handshake, and b) with an already open connection to measure the overhead of creating a new connection. All measurements are average values of multiple test runs on the *3dcube* Lambda with an additional warm-up phase for more stable results.

As can be seen in Tab. 2, keeping a connection alive is quite beneficial, as well as keeping and reusing contexts. Also, while all platforms have a similar overhead, *SecureDuk* has much higher latency as the actual Lambda processing by the Duktape JavaScript engine is slower. In most cases *SecureV8* has a slightly higher latency than the baseline except for new connections on a warm Lambda. This is due to the asynchronous system call processing of SGX-LKL that even outperforms the application outside the enclave, as there is a high number of system calls during the TLS handshake.

5.5 Conclusion of Evaluation

Our evaluation of the two platforms shows, that *SecureDuk* is by far a more lightweight platform with regards to the TCB, but *SecureV8* provides much better performance as it supports advanced features like JIT compilation. A more security-focussed cloud provider might want to prefer the more lightweight platform, as lower TCB usually implies less exploitable security vulnerabilities, while the *SecureV8* platform is clearly more suitable for high performance environments, therefore, a trade-off has to be made.

[12]h2load https://nghttp2.org/documentation/h2load.1.html

Table 2: Request latency (3D Cube Lambda).

	Baseline	***SecureV8***	***SecureDuk***
	New connection		
Cold	120.7ms	144.7ms	265.8ms
Warm	101.0ms	94.0ms	265.5ms
Overhead	93.6ms	76.4ms	93.0ms
	Open connection		
Cold	46.4ms	82.2ms	172.3ms
Warm	16.7ms	18.1ms	170.9ms
Overhead	0.9ms	0.9ms	1.0ms

6 RELATED WORK

Ryoan [16] allows users the processing of sensitive data in an untrusted cloud platform. The project builds upon Google's NaCl sandboxing mechanism to protect the OS from the content of the user-defined containers. This approach would be required if Lambdas contained native components or dependencies. However, this work focuses on providing a trusted FaaS and the aspect of efficient resource utilisation as well as the execution of JavaScript-based Lambdas as the common denominator in this domain.

Shen et al. [22] try to approach the inherent tension between isolation and sharing, especially in SGX applications, and propose a single-address-space solution comprising their library OS as well as all user-level applications in a single enclave. In their case, isolation is achieved by leveraging Intel MPX to support isolation between distrusting applications.

Boucher et al. [9] propose a FaaS architecture that focuses on small entities and language-based isolation, achieved by compiled micro services written in Rust. In contrast to their work, we focus on supporting legacy Lambdas without requiring them to be ported for the Lambda platform.

Alder et al. [4] propose a trusted FaaS architecture based on the Duktape JavaScript interpreter. Their work is orthogonal to this paper as they focus on the accounting of Lambdas in such a platform. Furthermore, this work focuses on the resource efficiency by comparing the use of two engines.

7 CONCLUSION

In this paper we proposed a novel design for a secure FaaS architecture that is tailored towards efficient resource usage by utilising the isolation capabilities of recent JavaScript engines. We validated our approach by two distinct implementations with different characteristics: the lightweight *SecureDuk* platform that is based on the Duktape engine and offers a relatively low TCB, and the *SecureV8* platform with much higher performance based on the Google V8 engine.

ACKNOWLEDGMENTS

This project has received funding by Intel Corp. in the scope of the *TFaaS* project.

REFERENCES

[1] 2017. Introducing Cloudflare Workers: Run JavaScript Service Workers at the Edge. https://blog.cloudflare.com/introducing-cloudflare-workers.

[2] 2018. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers.

[3] 2018. v8 dev: Untrusted code mitigations. https://v8.dev/docs/untrusted-code-mitigations.

[4] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2018. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. *arXiv preprint arXiv:1810.06080* (2018).

[5] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Hardware and Architectural Support for Security and Privacy (HASP).*

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[7] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[9] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the Micro Back in Microservice. *2018 USENIX Annual Technical Conference (USENIX ATC)* (2018).

[10] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[11] Stefan Brenner, Michael Behlendorf, and Rüdiger Kapitza. 2018. Trusted Execution, and the Impact of Security on Performance. In *3rd Workshop on System Software for Trusted Execution (SysTEX).*

[12] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference (Middleware).*

[13] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[14] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2016).

[15] Sanghyun Hong, Abhinav Srivastava, William Shambrook, and Tudor Dumitraș. 2018. Go Serverless: Securing Cloud via Serverless Design Patterns. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud).*

[16] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[17] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P).*

[18] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys).*

[19] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*.

[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security).*

[21] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Hardware and Architectural Support for Security and Privacy (HASP).*

[22] Youren Shen, Yu Chen, Kang Chen, Hongliang Tian, and Shoumeng Yan. [n. d.]. To Isolate, or to Share?: That is a Question for Intel SGX. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys).*

[23] Shweta Shinde, D Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Network and Distributed System Security Symposium (NDSS).*

[24] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference (ATC).*

[25] Raoul Van Bulck, Jo and Minkin, Marina and Weisse, Ofir and Genkin, Daniel and Kasikci, Baris and Piessens, Frank and Silberstein, Mark and Wenisch, Thomas F and Yarom, Yuval and Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security).*

[26] Tim Wagner. 2014. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/de/blogs/compute/container-reuse-in-lambda/.

[27] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC).*

[28] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2019. sgx-perf: Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference (Middleware).*

[29] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P).*