Atlas: Automated Scale-out of Trust-Oblivious Systems to Trusted Execution Environments

Anagnopoulos Georgios

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisors:
Assistant Prof. *Polyvios Pratikakis*,
Associate Prof. *Sotiris Ioannidis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

This work was partially supported by the European Project C4IIoT, funded by the European Commission under Grant Agreements No. 833828. This thesis reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Atlas: Automated Scale-out of Trust-Oblivious Systems to Trusted Execution Environments

Thesis submitted by
Georgios Anagnopoulos
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Georgios Anagnopoulos

Committee approvals:

Polyvios Pratikakis — Digitally signed by Polyvios Pratikakis Date: 2021.04.14 07:10:21 +03'00'

Polyvios Pratikakis
Assistant Professor, Thesis Supervisor, Committee Member

SOTIRIOS IOANNIDIS — Digitally signed by SOTIRIOS IOANNIDIS Date: 2021.04.04 12:18:12 +03'00'

Sotiris Ioannidis
Associate Professor, Thesis Supervisor, Committee Member

KONSTANTINOS MAGOUTIS — Digitally signed by KONSTANTINOS MAGOUTIS Date: 2021.04.04 13:50:14 +03'00'

Kostantinos Magoutis
Associate Professor, Committee Member

Departmental approval:

Polyvios Pratikakis — Digitally signed by Polyvios Pratikakis Date: 2021.04.14 07:10:40 +03'00'

Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, March 2021

Atlas: Automated Scale-out of Trust-Oblivious Systems to Trusted Execution Environments

Abstract

Trusted Execution Environments (TEEs) offer important security benefits to applications that combine on- and off-premise components. Acquiring these benefits, however, requires significant developer effort in order to identify and port security sensitive application components inside a TEE. The lack of high-level TEE APIs enforces the community to utilize low-level interfaces, commonly offered in C/C++, dealing with the complexities that low-level programming languages come with; *i.e.* memory handling, debugging and maintenance. Also, these security benefits come with a performance trade-off due to the added encryption/decryption schemes, integrity checking and protected memory limitations.

This work presents Atlas, a system for automatically scaling out components on TEEs, using a high-level programming language, namely JavaScript. Our system uses program transformations to offload the sensitive function calls of a given application and distribute the load among trusted nodes. This is achieved by embedding JavaScript's run-time environment within the TEE and performing the appropriate optimization in order to achieve remote function execution. We evaluate Atlas using a set of language-specific algorithms and cryptographic suites as well as with three real-world applications written in JavaScript. This results show that Atlas is able to scale-out legacy applications, originally not developed with TEE capabilities, with significant performance benefits. Our system is able to perform the execution up to 7 times faster compared to the vanilla QuickJS JavaScript interpreter, using ten TEE-enabled remote nodes, while also providing elasticity characteristics, all achieved with minimal developer effort.

ΑΤΛΑΣ: Αυτοματοποιημένη κλιμάκωση μη-ασφαλών συστημάτων σε Αξιόπιστα Περιβάλλοντα Εκτέλεσης

Περίληψη

Τα Αξιόπιστα Περιβάλλοντα Εκτέλεσης προσφέρουν αξιοσημείωτα οφέλη ασφαλείας σε εφαρμογές που είτε τα χρησιμοποιούν άμεσα είτε συνδυάζουν ένα σύνολο από τα στοιχεία τους. Αποκομίζοντας αυτά τα οφέλη, ωστόσο, απαιτεί προσπάθειες του προγραμματιστή για τον εντοπισμό και την διαμόρφωση των στοιχείων που χρείαζονται να εκτελεστούν σε αξιόπιστο περιβάλλον. Η έλλειψη εργαλείων για αξιόπιστα περιβάλλοντα γραμμένα σε γλώσσες προγραμματισμού υψηλού επιπέδου αναγκάζει τους προγραμματιστές να υλοποιούν τις εφαρμογές τους σε χαμηλό επίπεδο, αντιμετωπίζοντας παράλληλα και όλες τις δυσκολίες που συνοδεύουν αυτές τις γλώσσες, όπως ο προσεκτικός χειρισμός μνήμης, εντοπισμός σφαλμάτων ή συντήρηση των εργαλείων αυτών. Επιπλέον, η ενίσχυση της ασφάλειας της εκτέλεσης συμβάλλει στην μείωση της βέλτιστης απόδοσης, λόγω της προστιθέμενης κρυπτογράφησης/αποκρυπτογράφησης, δοκιμών ακεραιότητας ή περιορισμών δικαιωμάτων της μνήμης.

Αυτή η εργασία παρουσιάζει ένα σύστημα για την αυτόματη κλιμάκωση μη ασφαλών στοιχείων με τη χρήση γλώσσας προγραμματισμού υψηλού επιπέδου, JavaScript σε Αξιόπιστα Περιβάλλοντα Εκτέλεσης. Χρησιμοποιεί μετασχηματισμούς στο ίδιο το πρόγραμμα με σκοπό την εκφόρτωση και διανομή φορτίου μιας δεδομένης εφαρμογής μεταξύ αξιόπιστων κόμβων. Αυτή η λειτουργία επιτυγχάνεται με την ενσωμάτωση ενός συστήματος εκτέλεσης υψηλού επιπέδου γλώσσας προγραμματισμού σε αξιόπιστο περιβάλλον. Η αξιολόγηση πραγματοποιήθηκε σε ένα σύνολο αλγορίθμων βασισμένο στην ίδια την γλώσσα προγραμματισμού, μια κρυπτογραφική σουίτα αλλά και 3 πραγματικές εφαρμογές γραμμένες εξ ολοκλήρου σε JavaScript. Η αυτόματη κλιμάκωση των εφαρμογών δείχνει θετικά αποτελέσματα και σημαντικές επιταχύνσεις σε σχέση με την απλή ενίσχυση της εφαρμογής με μέσο όρο της τάξεως του $7times$ με σύστημα αποτελούμενο από 10 κόμβους.

Acknowledgments

I would like to express my deepest appreciation to my advisor Professor Sotiris Ioannidis, for assisting me throughout my academic career and inspiring me to conduct novel research as a member of the DiSCS laboratory. His wise guidance contributed to the amelioration of this long and hard-working process of earning my Master's Diploma.

Furthermore, I would like to thank Professor Polyvios Pratikakis for introducing me to the parallel computing field as well as for his eagerness to transfer his knowledge either through classes or personal interaction.

Last but not least, I want to thank my family and friends for wisely counseling me throughout the years and selflessly supporting me whenever needed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Trusted Execution Environments (TEEs), such as Intel SGX [1] and ARM TrustZone [2], provide significant security benefits to software executing on top of untrusted environments. Programs or program fragments executing in TEEs can provision enclaves that remain isolated from untrusted components, even when these components execute with higher privilege—*e.g.*, BIOS, drivers, operating systems, or hypervisors in shared multi-tenant infrastructures. These security benefits are pronounced in cases where programs make use of off-premise infrastructure for tackling scalability, availability, or fault tolerance concerns. In these cases, programs offload part of the computation to a public cloud vendors, such AWS [3], Azure [4], or GCloud [5], while maintaining certain integrity and confidentiality requirements. Offloading benefits include speeding up computations, mitigating resource-exhaustion attacks, improving availability and load balancing during spikes [6].

Unfortunately, the number of applications engineered as trust-aware systems, carefully designed from scratch to operate partly on a TEE environment, is quite small. Most applications are developed and deployed in a trust-oblivious fashion—that is, until the need to tackle offloading related security issues. When this happens, developers try to decompose the program and manually rewrite parts of it to leverage TEEs. The scope of such rewrites can vary considerably, typically requiring manually partitioning the application; *i.e.* distinguish the critical fragments from the rest. Furthermore, limiting the size of the Trusted Computing Base (TCB) to prevent Return Oriented Programming (ROP) attacks is oftentimes needed, as it has been shown that large TCBs contain more gadgets, increasing the possibility of this kind of attacks [7]. This effort is expensive and can introduce new bugs, cascading changes, or other incompatibilities, all compounded by today's extensive use of third-party libraries [8].

Could the process of decomposing a program, offloading part of its execution in a TEE and scaling it out at run-time be significantly automated? To answer this question, we present a series of techniques that, all combined, allow a trust-agnostic program to dynamically scale its execution over multiple TEE-enabled nodes —all with minimal-to-zero developer effort. We bring all these techniques together and develop Atlas, a system that allows secure TEE-assisted scaling-out of new and legacy JavaScript applications.

To allow delineating program fragments that are candidates for TEE scale-out, Atlas provides a structure with all the available package calls. Atlas packs a series of automated program transformations that operate at run-time to scale fragments over the available TEE-enabled resources. Atlas's run-time support completes the picture by providing a TEE-enabled language run-time, language-aware serialization support, remote module loading, and other system functions.

Our evaluation, applied to three real-world applications *i.e.* a Machine Learning toolset `ml.js` [9], a Natural Language Processing library named `Compromise` [10] and a simple password manager [11], shows that Atlas achieves speedups that range between 4–7× (avg: 5.5×) over the vanilla run-time. Moreover, their manual decomposition requires on average only 300 LoC of changes while Atlas is also able to offer significant speedup even to completely unmodified applications.

1.1 Contributions

The contributions of this work are:

- We design and implement Atlas, a distributed system leveraging Intel SGX that enables automatic offloading of JavaScript applications to confidential nodes, eliminating the need of porting code to device-specific TEEs.
- In order to accelerate Atlas's performance, we optimize the vanilla QuickJS implementation and develop additional scaling and load balancing features.
- We evaluate Atlas with a series of micro- and macro-benchmarks as well as with three real world applications. The results indicate that Atlas provides an average performance speedup of 5.5x to real-world applications, compared to the vanilla version.

1.2 Outline

The rest of this thesis is organized as followed. Chapter 2 presents the background on most popular Trusted Execution Environments currently available on the market. Furthermore, we explain why using a high-level programming language within a TEE is beneficial to users and how cloud computing can accelerate the overall performance of an application or system. In Chapter 3, we present the threat model of our system and an example that depicts the model. In Chapter 4 we describe extensively the main components of Atlas and its execution flow. On Chapter 5, we demonstrate the process of porting QuickJS within an SGX enclave and the challenges we faced. Also, we show the optimizations made in order to enhance the overall enclave's performance in order to minimize the introduced overheads. Chapter 6 presents an in-depth analysis and evaluation of Atlas with a variety of micro- and macro-benchmarks, including language-specific paradigms and three real world applications. In Chapter 7, we discuss the limitations of our system that we were able to successfully mitigate, as well as those that still limit our system. In Chapter 8

we examine existing work on TEE-enabled systems combined with code offloading and finally in Chapter 9 we analyze our future steps.

"All code is legacy. It's like how the speed of light is finite, which means technically you're always looking at the world as it was at some time in the past. You commit some new code, you go for a coffee break, you come back—it's legacy code now."

qntm

Chapter 2

Background

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a secure and isolated processing environment, providing secure memory and storage capabilities. Since no well-defined term of TEE existed by 2015, Mohamed Sab et.al [12] proposed a new definition for TEEs, taking into consideration all the previous research on trusted execution [13, 14, 15]. Quoting this definition: "Trusted Execution Environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the run-time states (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of its code, data and run-time states stored on a persistent memory."

2.2 Intel Software Guard Extensions

One of the most popular TEEs is Intel's Software Guard Extension (SGX) [1]. SGX is an ISA extension for protecting selected code and data from disclosure attacks or modifications. Intel SGX makes such protections possible through the use of `enclaves`, shielded environments for applications. Enclave code and data reside in Enclave Page Cache (EPC), which is a region of protected physical memory. Both enclave code and data are guarded by CPU access controls and are also cache-resident. Every time the data are moved to DRAM, they are encrypted via an dedicated on-chip memory encryption engine (MEE) at the granularity of cache lines. For Intel Skylake CPUs [16], the EPC size is between 64 MB and 128 MB and SGX provides a paging mechanism for swapping pages between the EPC and untrusted DRAM, introducing some additional performance overhead.

Enclave memory is also protected against memory modifications and rollbacks, using integrity checking. Non-enclave code cannot access enclave memory, however enclave code can access untrusted DRAM outside the EPC directly. However, the developers have to explicitly define a set of functions that may access the enclave from the unstrusted part, whose goal is to perform several trusted actions, called `ecalls` (enclave calls). Moreover , they have to define a set of unstrusted functions that reside in the non-secure

memory and may be called by the trusted part, named `ocalls` (outside calls). It is the responsibility of the enclave code, however, to verify the integrity of all untrusted data. Application code can be put into an enclave by special instructions and software made available to developers via the Intel SGX SDK. The Intel SGX SDK is a collection of APIs, libraries, documentation, sample source code, and tools that allows software developers to create and debug Intel SGX enabled applications in C and C++ and is targeted for x86_64 computer systems.

Shielded Application An SGX application can be viewed as two separate and distinct entities that communicate through a trusted channel. The untrusted part of the application is located and being executed in non-secure memory and has full access to the hardware resources available, whereas trusted code is executed within the enclave and the enclave page cache. SGX applications are executed strictly in Ring 3 (user-space) and use a stripped-down version of `libc`. System calls and requests for the enclave need to be defined in the EDL file before compilation. The tool responsible for this communication between these two worlds is `Edger8r`. Finally, the Enclave Signing Tool can be invoked to sign the executable. Any post changes to the source, data or the signature itself can be detected and force the abortion of the execution.

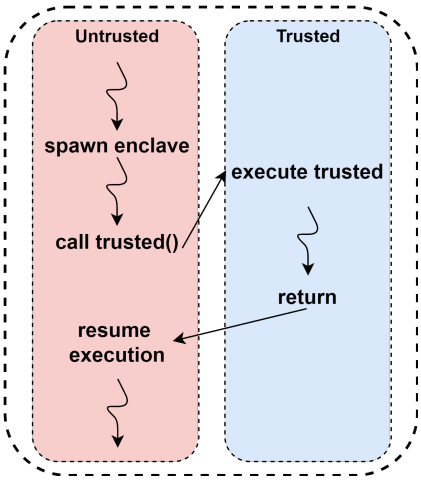


Figure 2.1: A trusted function execution

2.3 JavaScript

JavaScript (JS) is a popular high-level programming language that allows users to create multiple kinds of applications, varying from small command line applications to complex web-based services. According to Github analytics [17], JavaScript is the de facto web programming language globally and the most commonly used among developers. Another important feature is the module ecosystem, which amplifies the modular programming characteristics through the use of third party modules. However, since JavaScript is a dynamically typed language, meaning that a variable can change state and type at run-time, means that is also error-prone to bugs generated by the misuse of this feature.

One of the most prominent JavaScript standards is `ECMAScript`, which is a general-purpose programming language itself. ECMAScript is a standard for ensuring the interoperability among different engines while `ECMA262` [18] is its specification. Thus, JavaScript is an implementation of ECMAScript which conforms to the ECMA262 specification. The current version of ECMAScript is 6.0 and is also called `ES6` [19]. The most well-known JavaScript run-time environment is `NodeJS`[20], publicly used by many industries like LinkedIn, PayPal and Amazon Web Services. The underlying engine of `NodeJS` is called `V8` [21], a new generation of ECMAScript engines running Just-In-Time compilation, *i.e.* a way of executing code that includes compilation during the run-time execution of a program. NodeJS uses the `CommonJS` [22] module specification for working with modules, which was introduced to assist the server-side development of apps by reusing modules in divergent environments.

As *"JITTing"* is faster and more advanced way to execute a JavaScript program, many developers choose to run JavaScript using run-time interpreters. The most popular JIT engines are `Rhino` [23], build from Mozilla entirely in Java, `Futhark` or `Caracan` [24], the underlying engine of the Opera Browser [25] and QuickJS [26], a small and embeddable JavaScript engine written entirely in native C. In our system we chose the latter, as its small code-base (~2.5MB) and lack of JIT make its porting within SGX enclaves possible.

File Name	Size	File Name	Size
cutils.c	20K	libbf.h	20K
libbf.c	236K	libregexp.h	4.0K
libregexp.c	84K	libregexp-opcode.h	4.0K
libunicode.c	48K	libunicode.h	4.0K
qjs.c	20K	libunicode-table.h	208K
qjsc.c	24K	cutils.h	8.0K
quickjs.c	1.7M	quickjs-atom.h	8.0K
quickjs-libc.c	120K	quickjs.h	44K
server.c	4.0K	quickjs-libc.h	4.0K
unicode-gen.c	84K	quickjs-opcode.h	16K

Table 2.1: QuickJS file sizes

2.4 Cloud Computing

Nowadays, the volume of data has greatly increased and today's analytics often require distributed computing capabilities. Cloud Computing (CC)[27] and the Internet of Things (IoT) are emerging and are crowned as the new platforms of the current generation. There are several applications and use-cases of these platforms, ranging from Infrastructure-as-a-Service (IaaS), Hybrid Cloud Computing, Analytics, Storage and databases and health-care services. The main goal of all these appliances is to achieve faster innovation, control flexible resources, and economies of scale. Specifically, there are four types of cloud computing:

- **Infrastructure as a Service (IaaS)**: The service provider manages the physical hardware and supplies more computing resources if needed. The customer manages operating systems and applications.
- **Platform as a Service (PaaS)**: The service provider manages everything except application code and deployment to customers.
- **Software as a Service (SaaS)**: The service provider manages the entire chain up to the final application. The customer only has to put data into the application to get the service they need.
- **Function as a Service (FaaS)**: The service provider manages the function level, meaning that application code or containers can run partially on the service.

Typically, average users may only pay for cloud services they use, helping them lower their operating costs, run the infrastructure more efficiently, and scale-up as their business needs change. Some of the most vastly-used cloud-services are the Amazon Web Services Cloud (AWS)[3], the IBM Cloud [28] and the Google Cloud [5] and the offered services may vary. For example, the Amazon Web Services consist of two main entities: (i) Amazon Elastic Compute Cloud (EC2) and (ii) Simple Storage Service (S3). EC2 is a service where users can create virtual machines and run them on one of Amazon's data centers. These virtual machines take a fragment of the infrastructure and simulate the hardware of a physical server. CPU power, memory limit, virtual hard drive sizes and the Amazon Machine Image (*i.e.* operating system) are some of the configurable options when instantiating EC2. This cloud is really popular due to its processing power and immense bandwidth. AWS even supports compatibility with the Ethereum [29] blockchain and can provide node health, *i.e.* fault-tolerant nodes, and automatic software upgrades, improving the availability of the Ethereum infrastructure [30].

2.5 Motive

So combining the aforementioned knowledge, a question was raised:

"Can we trust the Cloud Provider?"

Chapter 3

Threat Model

Atlas assumes a powerful malicious party —*i.e.* the cloud provider, that has elevated privileges as well as physical access to the hardware but can not physically tamper with the CPU. Moreover, the adversary is able to manipulate the entire software stack, including the Operating System kernel. However, denial-of-service attacks (DoS) on key components, including but not limited to blacklisting the SGX or NIC drivers, preventing access to the file-system etc., are out of the scope of this work. Since a malicious party manipulating the OS is able to control an enclave's life cycle, we assume that they are able to prevent or abort the execution of the SGX enclaves or disrupt and eavesdrop network communications, However, they are yet not be able to gain any useful information by doing so.

Moreover, side-channel or other types of attacks targeting the Intel SGX hardware components, such as timing or page fault exploits, are orthogonal to Atlas and any commercial or research work that successfully manages to mitigate such attacks on SGX in the future can have a direct benefit to our system. Furthermore, Atlas assumes that the design and implementation of the Intel SGX SDK and its software stack is free of vulnerabilities. Finally, we assume that the implementation of the enclave-protected code as well as its untrusted driver-code is free of software vulnerabilities or bugs that could compromise it via remote input.

3.1 Threat Story

Now we will discuss the threat model using a simple story in order to explain the basic model of Atlas and how our system is able to provide secure computation in the cloud. The entities involved in this story and their roles are listed below.

Lets say Alice wants to execute a security-sensitive computation with confidential data in a fast and trustworthy manner. Her hardware capabilities limit her system's performance so she decides to offload the computation to the cloud. At this point, she questions whether she can trust the cloud provider. Then, she finds out Faythe (Atlas), a trusted cloud provider. Faythe encrypts all of Alice's data and operations and offloads them evenly across her powerful team of Teds. Eve comes into place and tries to listen,

Table 3.1: Characters

Names	Definition
Alice	The original main character.
Chuck	A third participant, usually of malicious intent.
Eve	An eavesdropper, passive attacker. She can listen in on messages but cannot modify them.
Faythe	A trusted advisor, courier or intermediary. Faythe is associated with faith and faithfulness.
Mallory	A malicious and active attacker who can modify, substitute and/or replay old messages.
Ted	A trusted arbitrator. In our case, Faythe's party.

but fails. When one Ted receives the message, Mallory and Chuck start to inspect Ted thoroughly. As all messages are decrypted within Ted's SGX enclave, neither of the malicious entities are able to understand what Alice says. The same procedure is followed for the result response path. Thus, Alice receives back the top-secret results, safe and sound.

"The only way to make software secure, reliable, and fast is to make it small."

Andrew S. Tanenbaum

Chapter 4

System Architecture

4.1 Atlas Overview

The overall architecture of Atlas is shown in Figure 4.1. Atlas consists of two main entities, (i) the Atlas library and (ii) the SGX workers residing on the cloud. The main library holds all the logic for wrapping the function calls along with all the metadata needed as well as the scheduler. Each computing node in the Atlas cloud is equipped with a QuickJS JavaScript interpreter, encapsulated within an SGX enclave itself; thus preventing any kind of unauthorized operation on the offloaded data.

4.2 Atlas Library

The Atlas client library is responsible for initiating the offloading operations to the remote SGX-enabled Atlas nodes. The library is implemented as a JavaScript module which the developers have to import in the script they wish to securely offload using Atlas. In this way, they are able to utilize the required functions that render the script or specific functions to be able to scale-out using Atlas. By designing the offloading mechanism as a thin library, the developers can perform the serialization and encryption of the source code and data to be offloaded, as well as perform the actual offloading to the trusted nodes, with minor script transformations.

4.3 Trusted Workers

The trusted workers are responsible for pre-loading all the available libraries and packages able to execute in our system during their initialization phase. Also, they are intentionally designed in such way that no arbitrary package execution can be invoked other than those supported by ES6 as well as our pre-built packages. When an enclave worker receives the encrypted data, it decrypts and deserialize them inside the secure SGX enclave in order to extract the name, arguments and package information. Throughout the execution, the offloaded code and data never reside in unprotected DRAM or storage in plain-text format.

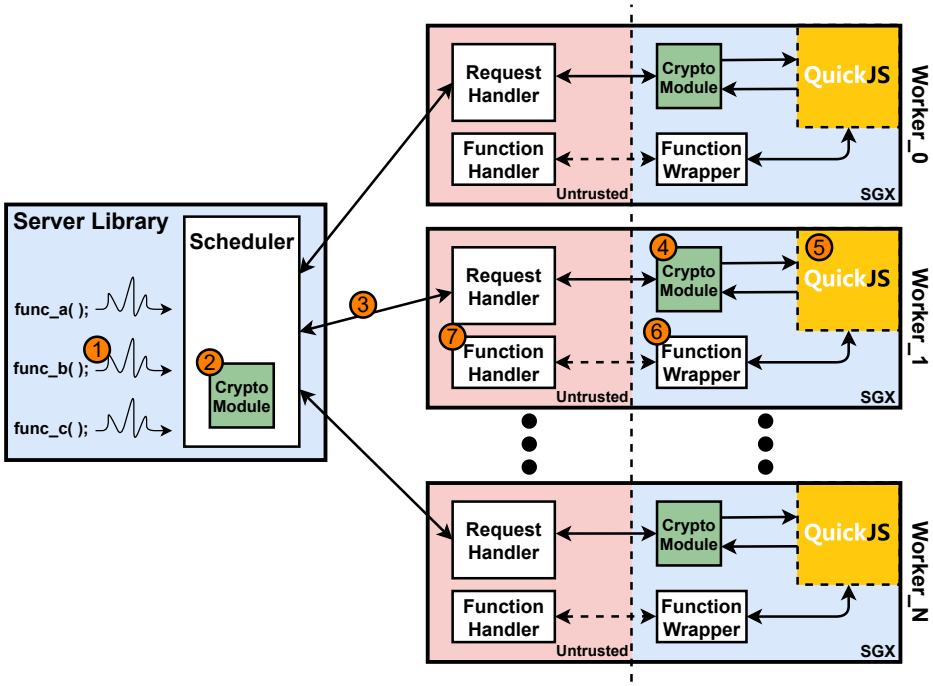


Figure 4.1: Atlas Overview of Atlas Architecture

4.4 Atlas Execution Flow

In this section, we present a typical Atlas execution flow, step-by-step. First, Atlas makes available a configuration file containing all the currently available SGX-enabled nodes in the form of IP and port pairs. Then, the Atlas library parses this file and sends a connection request to each one of the workers. After all the connections are established, the Atlas library performs a handshake with every worker using the Diffie-Hellman key exchange protocol After this point, the client, via the library, generates unique AES session keys for each worker establishes secure network connections. Then, the task offloading process can begin.

The next step requires the developer to define which functions have to be offloaded to the cloud. This is achieved by simply wrapping the respective sensitive functions with the `Atlas_wrap` function (1). At this point, the executable script is forwarded to the QuickJS interpreter and is analyzed. For each sensitive function encountered by the parser, the scheduler chooses an available worker in a round-robin fashion, packs a message with the function metadata and arguments, serializes, and encrypts the blob(2). Afterwards, the encrypted blob is forwarded to the remote worker (3). Once a worker receives the blob, it forwards it to its trusted region, *i.e.* . the SGX enclave, where the decryption takes place (4). When the blob is decrypted and deserialized, it is executed in the protected QuickJS interpreter residing in the enclave (5). In cases where system calls are needed for the proper script execution, they are forwarded to the trusted function wrapper (6) and then to the untrusted system call handler (7), responsible for serving them, and the results

return back to the protected QuickJS, inside the enclave. Once the worker completes the script's execution, the results are serialized and encrypted with the AES session key so that the data can securely be communicated back to the client. At the final step, the client gathers all the results from each worker and repeats the same procedure until all pending function calls are completed.

4.5 Working Example

We proceed with the demonstration of a simple Atlas use-case using pseudocode. First, the developer has to import our library along with the libraries required for the script's execution. Then, as shown in the fifth line of Figure 4.2, the developer wraps the untrusted object and then calls the sensitive function. This step corresponds to the first step depicted in Figure 4.1.

```
1 import obj from 'lib'
2 import atlas from 'atlas'
3 ...
4 ...
5 var wrapped = atlas.wrap(obj);
6 var result = wrapped.call(...args);
```

Figure 4.2: Client Source Example

When the object is wrapped, the library iterates each function call and wraps it with a new proxy function. In this way, when a function call is triggered, the Atlas main object starts the secure offloading process by serializing and encrypted (2) any information needed and sending it to one of the available workers in a round-robin fashion (3). A code snippet if this process is presented in Figure 4.3.

```
1 var stringified = atlas.stringify(...args);
2 var ciphertext = atlas.encrypt(stringified);
3 atlas.send(ciphertext);
4 atlas.recv(ciphertext);
5 var plaintext = atlas.decrypt(ciphertext);
6 return plaintext;
```

Figure 4.3: Server Source Example

```
1 worker.recv(ciphertext)
2 var plaintext = worker.decrypt(ciphertext)
3 var msg = worker.parse(plaintext)
4 var res = MOD_LIST[msg.func].apply(...msg.args)
5 var ciphertext = worker.encrypt(res)
6 worker.send(ciphertext);
```

Figure 4.4: Worker Source Example

Once the remote SGX-enabled worker receives the encrypted message it forwards it

to the enclave. The decryption takes place within the enclave as shown in the second line of Figure 4.4, which also corresponds to the 4th step of the Atlas execution flow, demonstrated in Figure 4.1. The worker then performs the deserialization of the blob and executes the requested function. Then, it encrypts the generated results and sends them back the to Atlas main object, as shown in lines 3-6 of Figure 4.4. Finally, the Atlas library decrypts the results and returns them to the client.

Chapter 5

System Implementation

In this section we are going to present the full implementation details of Atlas as well as the challenges that we encountered during the development and porting process and the optimizations we applied on the system in order to boost Atlas's overall performance.

5.1 Porting QuickJS

The JavaScript interpreter we choose to base Atlas on is QuickJS [26] which is developed in native C, making it a great candidate for our needs since SGX enclaves support only C/C++ applications. Also, its small code base, when compiled to binary, does not exceed the 128 MB live memory limit enforced by the enclaves, allowing for many JavaScript applications to be executed in the protected memory without triggering the expensive EPC page swap. Also, QuickJS is not based on JIT compilation, a feature that can not be supported by SGX.

Compiling the original source code of QuickJS generates two distinct binaries, the first one being the high level interpreter that parses and executes ECMAScript ES6 [19] while the second binary is capable of executing and compiling JavaScript code into native. We opt to utilize the first option since native module support developed in pure C, combined with QuickJS low level API, is not supported by our implementation as we describe in detail in Section 7.3). The original QuickJS sources contain 91K LoC of which 80K LoC are utilized by Atlas. Due to Intel SGX SDK limitations on supported functions and restricted system calls, we modify the interpreter's internal components and several parts of them are stripped, without breaking legacy script compatibility. As Intel SGX does not support certain features, we either implement them or strip them down, depending on their importance. Some of the features that are not needed, and thus removed, are: *bignumbers*,*atomics*, *signals* and *workers*, since Atlas does not support multithreading inside the enclave due to SGX's limitations. The bridging of QuickJS with the trusted component of Atlas is not a straight-forward task. First of all, several changes to the source tree have to be performed as SGX lacks system call and signal support while the I/O path and several other functions related to the execution environment need to be modifyed. Furthermore, we have to manually provide trusted bridges from the enclave

to the untrusted part of Atlas and vice-versa in order for the required system calls and functions that SGX SDK does not support inside the enclave to be executed. Finally, we provide dummy function implementations, within the enclave code, for functions required to be defined within QuickJS's code base, despite the fact that they are never utilized.

Apart from the trusted interpreter that is integrated inside the enclave, we also have to provide the necessary functionality in the trusted component of Atlas in order to enable it to act as a remote JS worker. This means that each distinct trusted Atlas entity has to set up an end-to-end secure communication channel with the client. After this point, our worker is ready to handle incoming execution requests from the client and forward the encrypted blob within the enclave where the decryption takes place using the exchanged secret `AES` key. Afterwards, the plaintext execution configuration is forwarded to. At this point, the QuickJS interpreter inside the enclave starts the script execution, retrieves the results, encrypts them using the same AES key and sends them back to the client by transferring the request to the untrusted part of the application.

5.2 Bootstrapping

Enclave initialization and creation is a costly operation. To ameliorate these unwanted costs that degrade the worker's performance, we choose to perform the following optimizations. First, we initialize the enclave and load the required libraries during the enclave creation time. Second, we choose to reuse the same enclave with the pre-loaded libraries for each new client offloading request, after having reset the enclave to a clean state. In this way, for each execution, we provide the enclave with all the required code, libraries and modules during the initialization phase and this process occurs only once. After this point, for each new client request, the only thing that has to be transferred to each independent Atlas worker is a configuration file in the form of `JSON`. This file contains information that points to the function that should be executed as well as its arguments. Pre-loading specific and limited libraries to our system does not enable the users to offload dynamically any code they need to executes but this process can be facilitated by altering the trusted module database, named `module_list`, by requesting a database update.

5.3 Distribution Steps

In order for a client to successfully offload and scale out the task execution in the cloud, several additional steps and processing layers are required.

5.3.1 Serialization

Prior to offloading the data to each distinct SGX worker, we have to serialize each client request. A typical client request contains information such as the library to be used, the target function to be executed, its required arguments and the node identifier. The arguments can be either plain data or complex data structures —*i.e.* graphs, hash-maps, arrays,

lists etc. To do so, we extend the functionality of the built-in JavaScript functions that QuickJS offers (`JSON_parse, JSON_stringify`) in order to support and serialize such data structures. Our custom parsing functions can be invoked with `Atlas.parse` and `Atlas.stringify`. Before a client offloads a request to each worker, it should first utilize the Atlas.stringify in order to serialize the request context before moving to the next step. Similarly, each different SGX worker has to invoke Atlas.parse to deserialize the data and then forward them to the interpreter.

5.3.2 Encryption

Since our main target is deploying and distributing tasks on cloud services, as mentioned in Section 3, we assume that the attacker has full control of the network interfaces and can monitor Atlas' traffic and connections. To overcome this issue, we add an extra encryption layer to the serialized data before sending them on the wire. Our main goal here is to offer fast encryption/decryption operations while achieving the best performance. For this reason, we extend our QuickJS implementation in the client component to support encryption and decryption in native C. This is done by invoking the high-level functions `Atlas.encrypt and Atlas.decrypt` which are linked with low-level OpenSSL implementations in native C. Finally, before offloading tasks to the workers, we invoke the encrypt function from the client side on the plaintext data and offload them to each one of the workers while the workers perform the decryption using the SGX built-in cryptographic functions. The decryption process at the worker side occurs strictly in the enclaves, hence no plaintext is leaked to the untrusted DRAM or file-system.

5.3.3 Networking

The most important part of job distribution at the client side can be divided in two separate steps: (i) establish the secure end-to-end communication, (ii) start sending encrypted request blobs to each remote worker. In order to initiate the connection to each one of the workers, we modify QuickJS to provide high-level functions, similar to the serialization functions described previously, in order to expose socket functionality to the JavaScript layer. We opt to follow this solution rather than using an available implementation since we the only functionality we need is to: (i) connect/close the sockets and (ii) send/receive, as we only need the bare minimum in order to achieve our goal. Thus, we provide basic socket functionality in low-level C and expose it using the functions `Atlas.connect, Atlas.close, Atlas.send and Atlas.recv`. After implementing this functionality, we can perform handshakes with all the available workers in our environment and exchange unique AES keys to encrypt the traffic. After this phase is completed and the keys are exchanged, the client can start to asynchronously offload tasks to each worker in a round-robin fashion.

5.4 Modules

In order to support module loading, the are two main properties that have to be supported inside the enclave. The first one is the `FILE` data structure in order to handle files and I/O. Secondly, all the functions that are invoked during the bootstrapping of QuickJS such as `fopen, fread, fwrite,` etc are re-implemented in order to fetch and load a QuickJS encrypted module from the untrusted file-system, decrypt it in the trusted part and verify its contents against a known checksum for integrity purposes. Finally, the interpreter is able to load the module and start executing it. This specific design prevents attackers from modifying or replacing modules found in the host system's memory/file-system or modules received over the network. The checksums required for module validation may be either pre-stored in Atlas's enclave, such as checksums for standard libraries, or received in encrypted format during the data transferring phase. With this mechanism in place, any `import` snippets found in a JavaScript script, import the declared library assuming that it exists in the untrusted file-system (encrypted) and then the module's API is exposed to the developer. This intermediate layer is transparent and offers the same functionality as the vanilla QuickJS module loading.

On the other hand, native C/C++ libraries are compiled as shared objectives using QuickJS's tool set and may only be loaded by Atlas's untrusted part, using the `dlopen` family of functions. This functionality prevents us from providing dynamic C/C++ library support for the following reasons. First, since shared objects cannot be loaded in Atlas's enclave, their integrity can not be verified by the system. Second, the functions loaded in Atlas's untrusted part may only interface with the enclave code via a predefined, at compilation phase, proxy layer (EDL). Thus, each function's prototype is not known during the execution of QuickJS code, something contradicting with the design of Intel SGX. Additionally, since the shared objects are stored in the file-system, and `dlopen` family of functions are not available in pure enclave code, a malicious entity may tamper and intercept their code.

5.5 Maintaining JavaScript State

One interesting aspect in distributed execution frameworks is the ability to preserve the execution context and state of each JavaScript environment. Assuming the scenario where we want to create and maintain a distributed password database, as described in Section 6.3.3, we do not want to clean the state after each request. Thus we can create it once and then keep re-using the same execution context so we can push, fetch and update its entries. Except from not resetting the enclave instance, we also do not reset the QuickJS execution context and run-time. However, keeping and re-using the same context might greatly increase the space complexity and introduce great performance overheads in the system. The solution to this problem can be achived with two steps: (i) de-allocate the unused/dead objects by assigning them to `null` without affecting the global functions and libraries, (ii) clean the run-time for each different execution context.

5.6 System-call Handling

The most common issues with many TEEs, including Intel SGX, is the lack of system call support. This is expected since in any real-life SGX-enabled scenario, only the underlying hardware and the SDK provided by the vendor are trusted by the system. So, system-calls, peripherals or even the OS kernel are considered untrusted and are excluded from the trusted environment. Thus, directly performing calls to the untrusted kernel, such as I/O calls or networking are not considered sound since a malicious user can intercept the calls and control/monitor the context and the data of each call. When utilizing Intel SGX, since such functionality is not available by Intel, everything has to be offloaded to the untrusted part of the application, where they are handled, using proxies (glue code that is declared in the EDL file). The results are then returned back to the enclave. Meanwhile, the enclave has no means of validating the results that it receives.

Normally, custom built SGX-enabled applications might require only a few system-calls that can be easily wrapped by the developers. However this scheme does not apply to Atlas. Since our implementation leverages and utilizes a full-fledged, drop-in replacement of QuickJS interpreter, enclosed in SGX enclaves, several challenges have to be addressed. First, we need to accommodate the system-calls that are required by the JavaScript virtual machine itself in order to function properly. Second, new system-calls may be issued during the execution of different scripts. The former case may be considered a more straightforward approach, as the required system-calls can easily be accounted for something that unfortunately does not apply in the latter case where all the system-calls requirements may be known a priori but will be resolved at execution time.

The most common approach to handle such an issue is to implement custom system-call wrappers for each one of them. However, modern operating systems provide several hundreds for system-calls (about 400 on Linux), without the majority of them being necessary required by the QuickJS interpreter or an executed script. Furthermore, many of these system-calls could be triggered and abused by an offloaded script to perform malicious activities on the remote host. For this reason, we decide to provide interfaces only for the bare minimum of the system-calls that are required in order to achieve the normal functionality of the interpreter inside the enclave but also to minimize the potential attack surface without having to keep proxying functions to the untrusted application. By observing the original source code, we realize that the most commonly used functions, resulting to system-calls, are those that are associated with I/O such as `fopen`, `fwrite`, `fread`, etc. In order to provide system-call support to the JavaScript interpreter residing within the SGX enclaves, we have to develop custom functions that will be defined in the EDL file and implemented in the non-enclave part of the application, performing an OCALL for each pending request. Once such a request is handled by the OCALL function, since the untrusted part of the application has full access to the whole system, the handle will get the results and forward them back to enclave after performing the typical SGX checks on the data. However, the validity of the data transferred from outside of the enclave within the enclave have to be explicitly checked for their integrity by the application developer. In the SGX model, the only entity we assume trusted is the enclave while the JavaScript scripts, modules and input data that are required for the execution must be provided with

each client request in encrypted format. In this way we prevent read and write operations to arbitrary file-system locations at the server side. Optional file offloading and handling may be performed by the client in predefined server file-system locations in an encrypted manner, using SGX's sealing and unsealing functionality offered by Intel's API, enabling secure and persistent storage in the untrusted file-system.

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."

Melvin E. Conway

Chapter 6

Evaluation

In this section, we present a thorough evaluation of our system. First, we evaluate Atlas with custom designed synthetic benchmarks that aim to stress the memory and I/O capabilities. Then, we measure Atlas's performance when executing a series of benchmark suites containing a variety of cryptographic and sorting algorithms as well as a suite of 15 popular algorithms. Finally, we evaluate Atlas using the following three popular real-world applications:

1. A Machine Learning toolkit named `ml.js` [9]
2. A small Natural Language Processing (NLP) library called Compromise [10]
3. A simple Password Manager [11]

Evaluation Setup The evaluation test-bed consists of the Atlas server, executing the client library, and ten remote SGX-enabled nodes, each one executing a Atlas worker. Each hardware node's specification is presented in Table 6.1. Every Atlas worker is compiled in hardware/signed SGX mode, preventing any debugging or enclave monitoring.

Table 6.1: Hardware Specifications of Atlas

Type	CPU Model	Base Clock	RAM	Kernel Version
Atlas server	Intel(R) Xeon(R) E5-2697	2.70GHz	32GB	Arch Linux 5.9.8
SGX Node	Intel(R) Core(TM) i7-6700	3.40GHz	16GB	Arch Linux 5.9.2

6.1 Micro Benchmarks

Data Retrieval In order to understand the impact of retrieving data from the untrusted environment in the secure enclave, we develop a simple JS script that reads a 20MB file from the file-system in chunks ranging from 64B up to 4MB. Then, we execute the script using both the vanilla QuickJS interpreter and Atlas using a single node and report

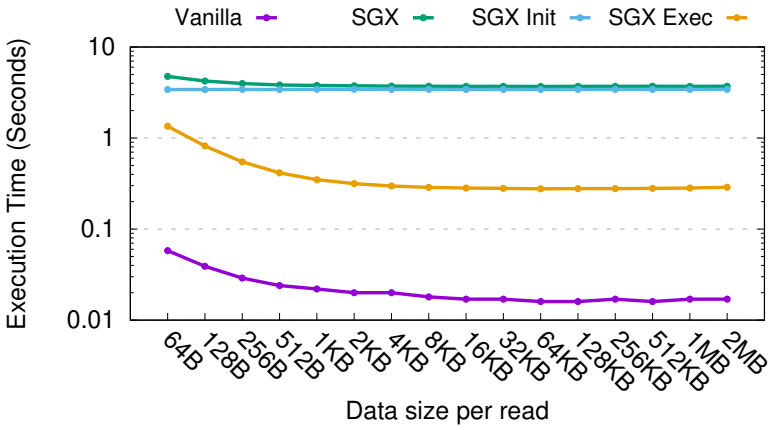


Figure 6.1: Fread performance

the results in Figure 6.1. We mark the execution time required by the vanilla QuickJS interpreter as *Vanilla* and the end-to-end time required by Atlas's SGX enclave as *SGX*. Furthermore, we present the time required for booting our system as *SGX Init* and finally, *SGX Exec* marks the actual execution time of the JS script inside the enclave. We notice that increasing the read data buffer significantly reduces the script's execution time as fewer system calls are involved and the execution is transferred fewer times between the enclave and the unprotected environment of Atlas. However, the encryption process for the enclave inbound data is responsible for increasing the execution time by an order of magnitude. Also, we notice that increasing the read data buffer beyond 2KB has little to no effect in further increasing the performance. Also, the overall end-to-end execution time of Atlas is increased compared to the vanilla QuickJS interpreter due to Atlas's boot time.

Memory Accesses We proceed the evaluation of the enclave's performance properties by measuring the overhead introduced by SGX when randomly accessing the protected memory with and without triggering enclave page swapping. For this reason, we develop a simple JS script that performs one million random writes to consecutive protected memory spaces, ranging from 64B to 2MB and afterwards one million random reads to the same locations. The results of this analysis are presented in Figure 6.2. As we can see, the memory access times achieved by Atlas, both for read and write operations, are almost identical to those achieved by the vanilla QuickJS interpreter when enclave page swapping is not triggered and the target memory space resides in the live protected memory area. Also, we notice that page swapping affects the overall execution of both interpreters, enforcing a slightly higher linear performance overhead for Atlas as the amount of non-live protected memory increases, since enclave page swapping involves cryptographic operations.

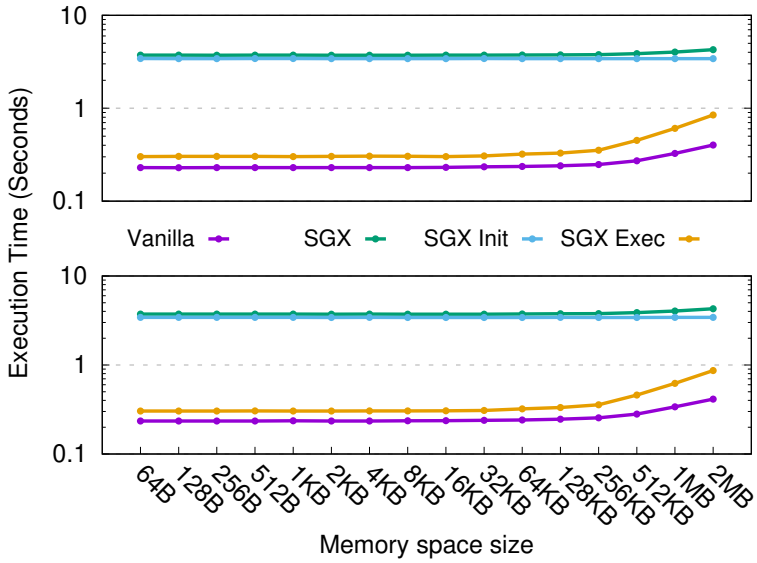


Figure 6.2: Memory read/write performance

QuickJS Memory Usage The next step is to measure the memory footprint of the QuickJS interpreter. In order to achieve this, we execute a plain JavaScript script and measure the memory usage and object count using the `JS_ComputeMemoryUsage()` and `JS_DumpMemoryUsage()` functions. In this way, we are able to get a view of QuickJS's initialization memory footprint. The results of this analysis are presented in Table 6.2. As we can see in the table, QuickJS has very minimal memory requirements, allocating only 23KB of memory and 2975 objects, with each object sized at an average of 8B. Totally, the interpreter uses 305KB of memory and 3203 objects. These characteristics render QuickJS an ideal interpreter to be encapsulated within SGX enclaves as its minimal memory requirements do not violate the 128MB protected live memory restriction and provides enough headroom for script execution without triggering SGX's expensive page swaping mechanism.

Table 6.2: QuickJS interpreter memory usage and object count for initialization and an empty script execution

Name	Count	Size (KB)	Bytes/Obj.
memory allocated	2975	23	8.0
memory used	3203	305	
strings	36	7	18.7
objects	407	29	72.0
properties	1778	31	4.4
bytecode functions	241	141	
C functions	98		
elements	116	2	4.8

6.2 Macro Benchmarks

6.2.1 Cryptographic Algorithms

The next step after analysing the properties that affect Atlas's performance is to proceed with the evaluation with a series of benchmark applications. We start this analysis by executing the SHA1 hashing algorithm for a 800K input repeatedly in order to specify a well-suited number of nodes for the Atlas cloud. While executing the calls needed, we increase the number of available nodes at run-time in order to observe the performance increase. As we can see in Figure 6.3, we gain speedup while increasing the number of nodes, with the optimal speedup achieved with ten nodes. We continue to increase this number by adding a second Atlas worker on every node but we notice a performance degradation as the memory needed for this execution exceeded the 128MB upper limit.

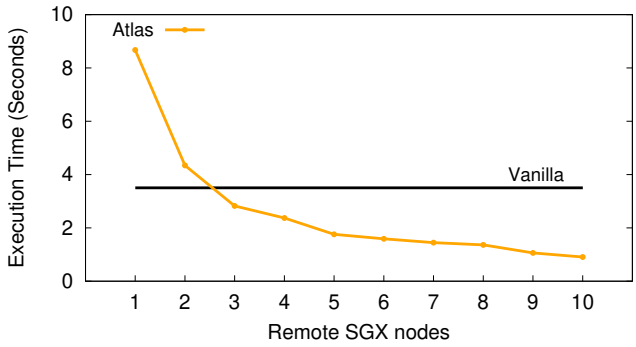


Figure 6.3: Sustained performance achieved by Atlas when executing the SHA1 hashing algorithm with various Atlas node configurations

Once we identify the optimal number of nodes for our hardware configuration, we evaluate Atlas with 8 popular cryptographic algorithms obtained from `crypto-js` [31]. We choose 6 popular hashing algorithms and two ciphers, namely AES and the Rabbit stream cipher. The evaluation is performed using the vanilla QuickJS interpreter and Atlas with one and ten nodes. Each algorithm is executed 10 times with an 800KB input and we present the average required execution time in Figure 6.4. Moreover, we present the speedup gained by utilizing Atlas on top of each bar. We notice that Atlas with one remote node has a performance overhead of 0.58× compared to the vanilla interpreter since Atlas has to transfer the scripts and the data to the remote host over an encrypted network channel as well as perform the execution inside the SGX enclave. The minimum overhead is reported at 0.8× for SHA3 and SHA384 while the maximum of 0.4× is introduced by SHA1 and MD5. However, when Atlas utilizes 10 remote nodes is able to significantly outperform the vanilla QuickJS interpreter while providing increased security. We notice that in this setup, Atlas provides an average speedup of 6.5× with the most performance bonus gained by SHA512 being 7.9× faster. The least performance gain is reported by MD5 executing 3.8× faster than on the vanilla interpreter.

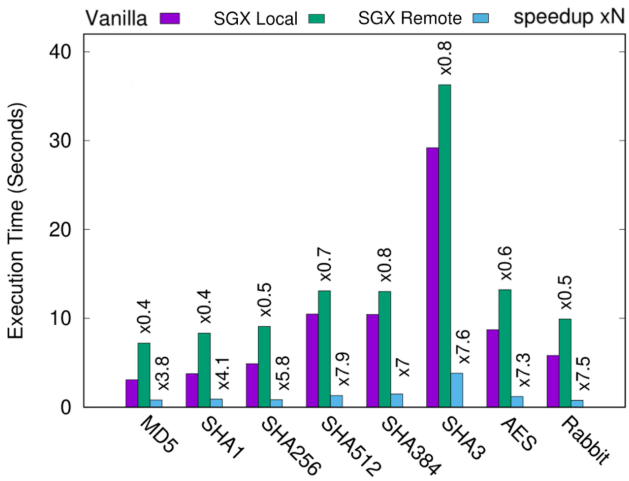


Figure 6.4: Cryptographic algorithms performance

6.2.2 Sorting Algorithms

We proceed our system's evaluation, this time using a set of ten popular sorting algorithms. We choose to evaluate Atlas with a series of sorting algorithms since they have different memory and computation needs and perform multiple memory accesses, each one with a different access pattern, while operating on the same data. In a similar fashion, we execute each algorithm 10 times, using the vanilla QuickJS interpreter and Atlas, and present the results in Figure 6.5. Again, we notice that Atlas with a single node has an average performance overhead of 0.67× with the highest reported at 0.3× for Bucket Sort and Counting Sort. On the other hand, Atlas with ten nodes is on average 7.2× faster than the vanilla interpreter, with the highest speedup being 9.2× for Gnome and the lowest 1.3× for Bucket Sort.

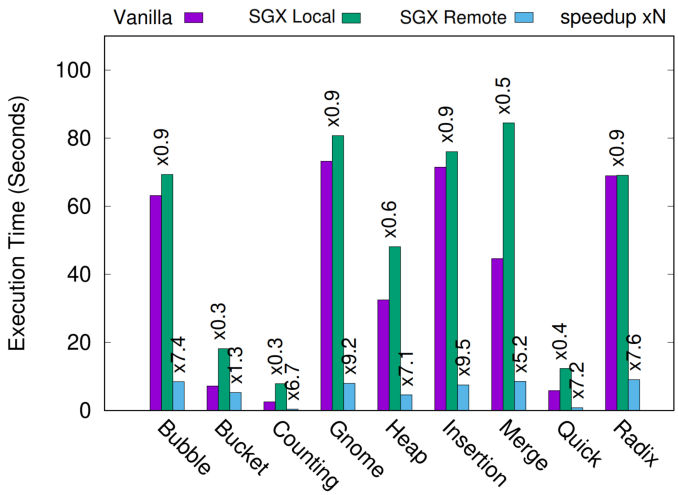


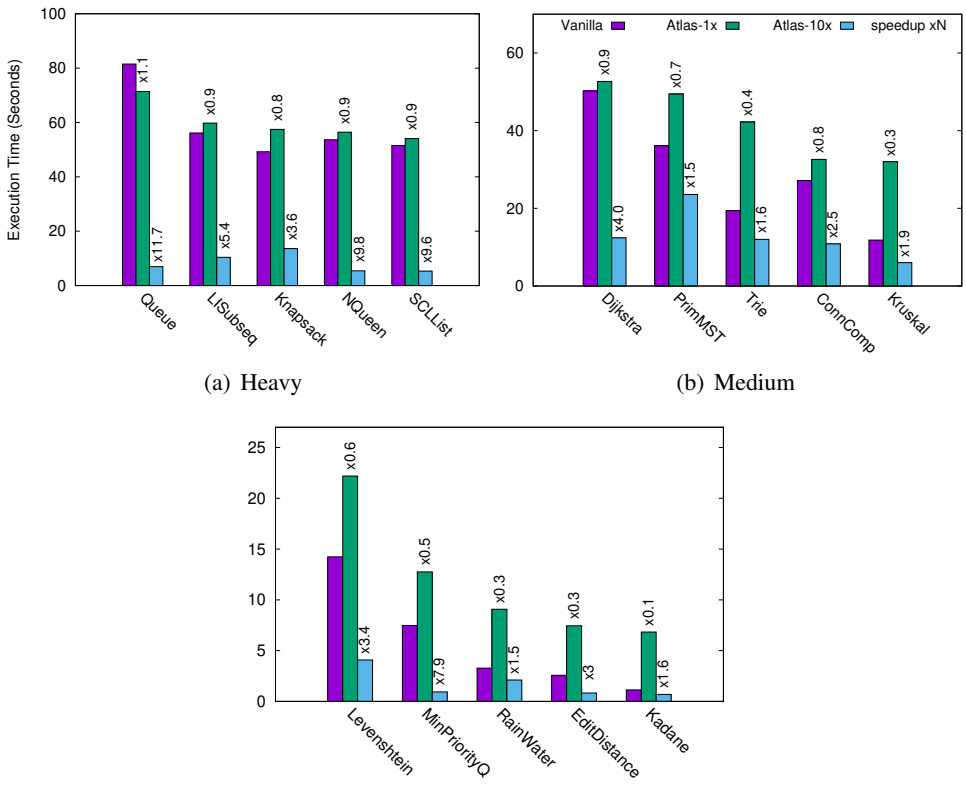
Figure 6.5: Sorting algorithms performance

6.2.3 Atlas Suite

In this section, we evaluate Atlas using a suite of 15 popular algorithms, found in many JavaScript applications, designed as benchmark applications. The selected algorithms provide a representative combination of complexity and memory resource utilization and each one operates on different data. This evaluation provides us with better insight on Atlas's behaviour when executing real-life applications. We execute each benchmark with the same setup, as described in the previous sections, and report the results in Figure 6.6.

As we can see, when utilizing a single node, Atlas introduces an average overhead of 0.63× with the maximum reported at 0.1× for Kadane's algorithm and the minimum at 1.1× for the Queue benchmark. We believe that this slight speedup is achieved since Queue is a very simple benchmark with low complexity that does not require dynamic memory allocations and is able to benefit from Atlas's optimized QuickJS interpreter implementation.

Finally, we can see that when Atlas utilizes ten nodes is able to execute the benchmarks on average 4.6× faster than the vanilla QuickJS interpreter. The highest reported speedup is 11.7×, gained by the Queue algorithm and the lowest is 1.5×, gained by Prim's Minimum Spanning Tree and the Trapping Rain Water algorithms.



(a) Heavy (b) Medium

(c) Light

Figure 6.6: Atlas suite sustained performance

6.3 Real World Applications

6.3.1 Application 1: Machine Learning

In this section we evaluate a series of machine learning algorithms found in `ml.js` [9], which is a machine learning toolkit, developed in native JavaScript and utilized by many real world applications. Our selected set includes both supervised and unsupervised learning algorithms as well as simple regression paradigms.

Evaluation Process We execute each algorithm using the vanilla QuickJS interpreter and Atlas in two modes, with one and ten remote nodes, connected to the server over an encrypted network channel. In order to be able to observe the benefit that Atlas's offloading provides, we carefully design each algorithm's input so as the required execution time exceeds the five seconds. The obtained results are the average of ten executions.

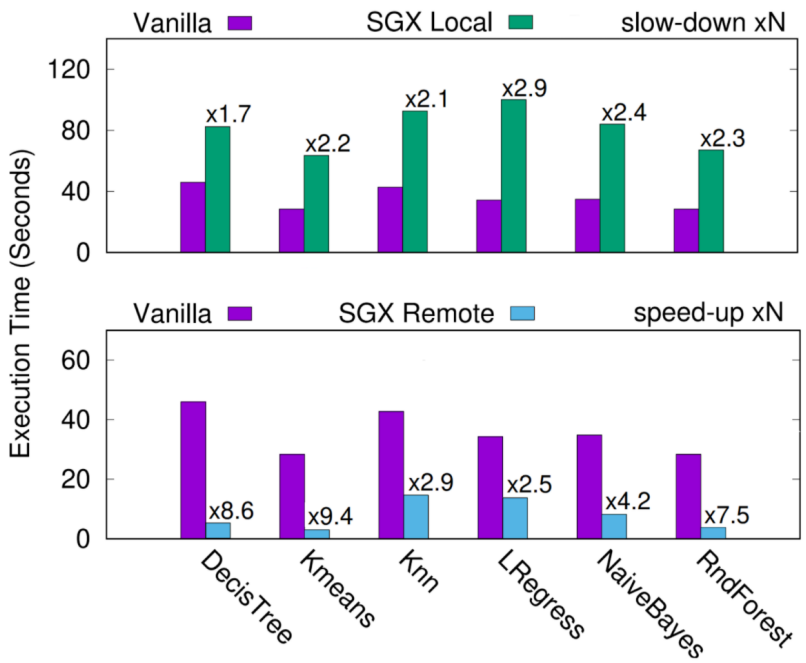


Figure 6.7: Machine-Learning algorithms performance

Results We report the results of this analysis in Figure 6.7 and present the reported speedup on top of each bar. Once again, without offloading the execution to multiple nodes, Atlas introduces an average overhead of 0.65× compared to standard QuickJS execution with the maximum being 0.4× for the Linear Regression and the minimum 0.8× for the Decision Tree algorithm. However, every algorithm is benefited by Atlas's scalability since the average speedup provided by Atlas when utilizing ten nodes is 5.83×.

We can see that K-means completes 9.4× with Atlas and the Linear Regression has the lowest speedup of 2.4×.

6.3.2 Application 2: Natural Language Processing

As a second real-world application, we choose the Compromise [10] Natural Language Processing library. Compromise is fast, compared to other NLP libraries of its category, and has a minimal code base of 375KB.

Evaluation Process In order to evaluate Compromise, we use the well-known comedy "Ornithes" (*i.e.* "Birds") by the ancient Greek playwright Aristophanes. The input, when converted to simple text, is approximately 200KB. Using the library, we develop an application that first finds all the (i) *verbs*, (ii) *nouns*, (iii) *adjectives* and (iv) *adverbs* in the input. Then it transforms all the *verbs* to past and present tense. Finally, the application transforms all the *nouns* to singular and plural form. We execute the application 10 times, using the vanilla QuickJS interpreter and Atlas, and report the average execution time results.

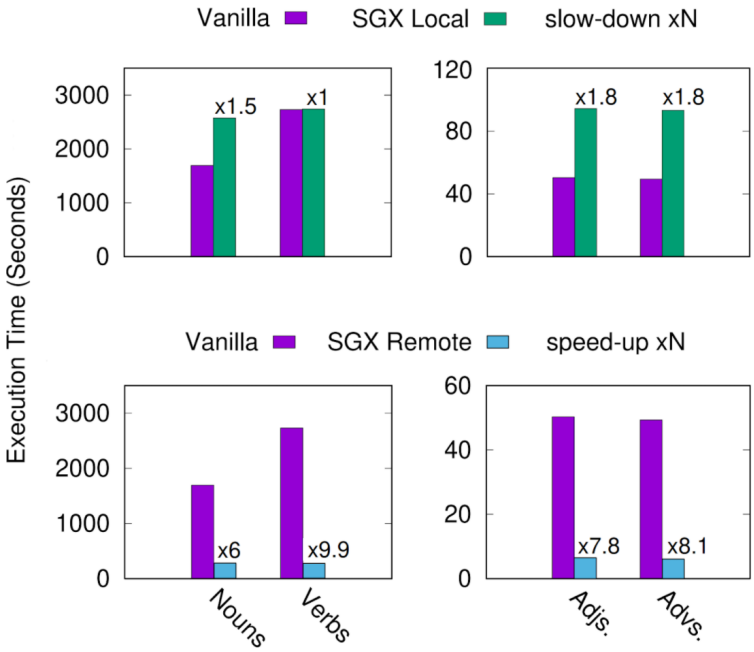


Figure 6.8: Natural Language Processing performance

Results The evaluation results obtained after the application's execution are presented in Figure 6.8. As we can see in the figure, the execution time required to process the *nouns* and *verbs* is two orders of magnitude higher than the ones required to process the

adjectives and *adverbs* due to the extra transformations performed. The average overhead introduced by Atlas in single-node mode is is 0.77× with the highest reported at 0.6× while locating the *nouns*. The process of finding the *verbs* executes slightly faster with Atlas and displays a speedup of 1.1×. On the other hand, when Atlas offloads the execution to 10 remote nodes is again able to provide an average speedup of 7.95×. The fastest process is locating and transforming all the *verbs* which gains a maximum speedup of 9.9× while the lowest speedup of 6× is achieved when processing the *nouns*.

6.3.3 Application 3: Password Manager

For our last real-world application we choose a simple open-source password manager [11].

Evaluation Process We modify the original script and develop an application that starts its execution by creating a database of 20K entries. Afterwards, it performs three types of batch operations on database, (i) *update password*, (ii) *read entry* and (iii) *delete entry*. Each one of these operations performs 200K transactions of each category, with each function call, in batches of 10k translations per batch. We carefully plan the requested transactions so as every remote node has served a *create db* operation as well as each *update* and *delete* is requested corresponds to valid entries. We execute the application 10 times, using Atlas and the vanilla interpreter, and record the average execution times.

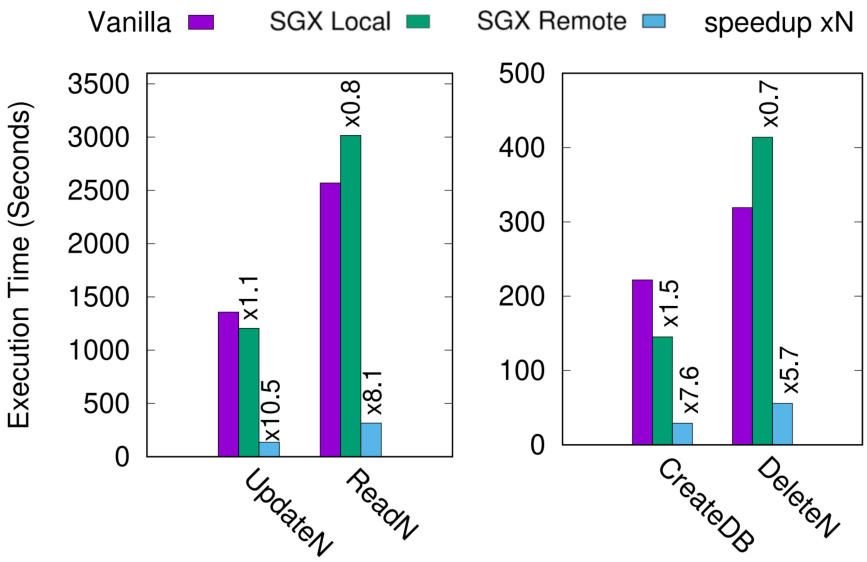


Figure 6.9: Password Manager performance

Results The results after executing the application with the vanilla QuickJS interpreter and Atlas are presented in Figure 6.9. We notice that the *update* and *read* operations yield

a significantly higher execution time than the one required by the *create db* and *delete* operations. This is explained by the fact that the *read* operations result in a returned result every time, stressing the I/O path between the Atlas server and nodes. Also, the *update* operations trigger memory writes in the protected memory, each time. Atlas's average overhead when executed with one node is $1\times$, with the maximum reported at $0.7\times$ for the *delete* operation and the minimum at $1.5\times$ for the *create db* operation. By examining the results, we see that in the case of the password manager, Atlas does not introduce any overhead at all, even in single-node mode. Also, two operations, namely *update* and *create db* gain speedups of $1.1\times$ and $1.5\times$ respectively. Finally, as expected, Atlas in ten-node mode provides an average speedup of $7.95\times$. The *update* operation is the most benefited with a performance gain of $10.5\times$ while the lowest speedup is reported $5.7\times$ for the *delete* operation.

Chapter 7

Limitations

7.1 Local SGX Execution

An alternative approach to Atlas's remote code execution model would be to spawn all the workers in the same physical host. However executing and offloading code on the same physical host introduces several security concerns. First of all, in case our physical host is trusted, we can indeed offload code locally without the need for encryption but the overhead of spawning several distinct enclave applications, containing independent interpreters, would degrade the overall performance. Furthermore, the live protected memory limitation of SGX, which is 128MB, will cap so fast that will constantly trigger the expensive page swapping mechanism. In the second case, a malicious user may observe the memory fragments of the executed code and steal the encryption keys, resulting in leaking the code executing inside the enclave along with its respective inputs.

7.1.1 SGX enclave constraints

As mentioned in the previous section, Intel SGX has a live protected memory limitation of 128MB. Exceeding this memory limit results in triggering the default page swap mechanism in Linux OS. Then, Intel's Memory Encryption Engine will start encrypting the trusted pages and storing them to the untrusted RAM before fetching and decrypting new ones. Consequently, the enclave developer has to carefully choose the maximum enclave memory that is required for the current enclave. As a result, applications that require several hundreds MB of live memory cat execute on Atlas but might suffer from noticeable performance overheads.

7.2 CommonJS

One major concern in our implementation revolves around JavaScript module support. NodeJS is the most popular JavaScript interpreter and the majority of JavaScript modules are developed on CommonJS. As noted in Section 5.1, QuickJS is based on ECMA6, thus

loading JavaScript modules developed in CommonJS standard is not enabled. Our approach, dynamically loading and executing CommonJS modules, requires a lot of manual effort in order to port and import the existing code.

7.3 Native module support

As described in Section 5.1, native module support is not enabled on Atlas. Dynamically loading and executing shared object libraries requires the use of the `dlopen` family of functions. Since these functions need to dynamically allocate and map the memory as executable they are not supported as this functionality is strictly prohibited and is not currently supported by SGX version 1. By offloading such unsupported requests to the untrusted component of the Atlas worker, we can indeed enable and execute shared libraries. The problem here lies in the execution of `dlopen` function since it is a system call and it has to be handled by the OS kernel. Furthermore, the mapping process will take place in the untrusted DRAM, resulting in exposing the dynamic library to a potential attacker that can scan the whole memory.

"We shall do a much better programming job, provided we approach the task with a full appreciation of its tremendous difficulty, provided that we respect the intrinsic limitations of the human mind and approach the task as very humble programmers."

A. Turing

Chapter 8

Related Work

8.1 Trusted Execution Work

Haven [32] executes unmodified legacy Windows applications inside SGX enclaves using a ported Windows libOS within SGX. Also, Graphene-SGX [33] encapsulates the entirety of the libOS as well as a trusted run-time with a customized C library and ELF loader inside an SGX enclave. SCONE [34] is a shielded execution framework that enables developers to compile their C applications into Docker containers. These works provide a container-based approach in which users can execute a vast variety of applications. Glamdring [35] is a framework that partitions and secures applications written in native C. The developer explicitly pinpoints the crucial data using language-based annotations. Similar to Atlas, Civet [36] is a framework which partitions Java applications and embeds fragments within the SGX enclaves. This system also provides garbage collection with an execution performance trade-off. TrustJS [37] also examines the feasibility of connecting JavaScript browser applications with SGX enclaves. However, it does not attempt to provide an environment for legacy JavaScript applications. RUST-SGX [38] is build atop of SGX in order to tackle unsafe components with proven memory safety and exterminating memory corruption inside SGX enclaves.

Moreover, several works aim to improve SGX's security properties, such as SGX-Shield [39] and T-SGX [40]. SGX-Shield [39] performs Address Space Layout Randomization (ASLR) in enclaves, with its goal being to maximize the entropy hide and enforce ASLR decisions. T-SGX [40] combines SGX with Transactional Synchronization Extensions in order to tame controlled-channel attacks. The aforementioned works are orthogonal to our implementation and can be directly integrated into Atlas in order to improve its security characteristics.

8.2 Distributed Execution Work

In 2020, Wang et al. [38] proposed an architecture that offers secure migration of legacy industrial control systems to the cloud and investigate whether the cloud can meet the real-time requirements that control operations demand, without compromising system safety.

In the previous year, the same authors also introduced a distributed fine-grained access control mechanism with computations offloaded on an IoT cloud [41]. CODE-FS [42] is also based on the same general idea as Atlas; partitioning and encrypting the code and all the metadata required for distributed computing. Finally, SAFE [43] is an architecture that also leverages secure distributed execution using Trusted Execution Environments.

Chapter 9

Conclusion

Summarizing this work, we propose and design Atlas, a distributed system, build with Intel SGX support, that enables automatic offloading of JavaScript applications to secure remote nodes, eliminating the need of porting new or legacy code to device-specific TEEs.

In order to accelerate Atlas's performance, we optimize and implement additional features for the QuickJS JavaScript run-time environment. Then, we examine how Atlas can be used by a vast set of language-specific applications, cryptographic suites and three popular real-world applications. Our evaluation shows that Atlas is able to offer an average performance speedup of up to 7 times to real-world applications, compared to the vanilla implementation, while protecting the offloaded code and data with secure SGX enclaves.

Bibliography

[1] "Intel Software Guard Extensions (Intel SGX)," https://software.intel.com/en-us/sgx.

[2] "ARM Security Technology, Building a Secure System using TrustZone Technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[3] "Amazon Web Services," https://aws.amazon.com/.

[4] "Microsoft Azure: Cloud Computing Services," https://azure.microsoft.com/.

[5] "Google Cloud: Cloud Computing Services," https://cloud.google.com/.

[6] G. Giakoumakis, E. Papadogiannaki, G. Vasiliadis, and S. Ioannidis, "Pythia: Scheduling of concurrent network packet processing applications on heterogeneous devices," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2020, pp. 145–149.

[7] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 523–539.

[8] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization." in *NDSS*, 2018.

[9] "Machine learning tools in JavaScript," https://github.com/mljs/ml.

[10] "Modest nNatural Language Processing," https://github.com/spencermountain/compromise.

[11] "A Simple Javascript Password Manager," https://github.com/andyy5/simple-pw.

[12] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.

[13] "Introduction to Trusted Execution Environments," https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 193–206.

[15] "ADVANCED TRUSTED ENVIRONMENT: OMTP TR1," http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf.

[16] "Intel's Skylake Processors," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf.

[17] https://madnight.github.io/githut/#/pull_requests/2020/4.

[18] "ECMAScript® 2021 Language Specification," https://tc39.es/ecma262/.

[19] B. Eich, "ECMAScript 6," http://es6-features.org/.

[20] https://nodejs.org/.

[21] https://v8.dev/.

[22] "Common JS," https://flaviocopes.com/commonjs/.

[23] https://github.com/mozilla/rhino.

[24] https://dev.opera.com/blog/carakan/.

[25] https://dev.opera.com/.

[26] F. Bellard, "QuickJs Javascript Engine," https://bellard.org/quickjs/.

[27] IBM, "Top 7 Most Common Uses of Cloud Computing," https://www.ibm.com/cloud/blog/top-7-most-common-uses-of-cloud-computing.

[28] ——, "IBM Cloud," https://www.ibm.com/cloud.

[29] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[30] "AWS Etherium Support," https://aws.amazon.com/about-aws/whats-new/2021/03/announcing-general-availability-of-ethereum-on-amazon-managed-blockchain/.

[31] "JavaScript Library of Crypto Standards." https://github.com/brix/crypto-js.

[32] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.

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

[34] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 689–703.

[35] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for intel {SGX}," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 285–298.

[36] C.-C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, "Civet: An efficient java partitioning framework for hardware enclaves," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[37] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "Trustjs: Trusted client-side execution of javascript," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[38] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards memory safe enclave programming with rust-sgx," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2333–2350.

[39] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.

[40] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." in *NDSS*, 2017.

[41] Q. Wang, G. Lv, and X. Sun, "Distributed access control with outsourced computation in fog computing," in *2019 Chinese Control And Decision Conference (CCDC)*. IEEE, 2019, pp. 2446–2450.

[42] M. Manek, A. Chhadwa, K. Shah, M. Potey, and M. Khan, "Cloud oriented distributed and encrypted file storage (code-fs)," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–5.

[43] J. Buchmann, G. Dessouky, T. Frassetto, Á. Kiss, A.-R. Sadeghi, T. Schneider, G. Traverso, and S. Zeitouni, "Safe: A secure and efficient long-term distributed storage system," in *Proceedings of the 8th International Workshop on Security in Blockchain and Cloud Computing*, 2020, pp. 8–13.

[44] "Algorithms in JavaScript," https://github.com/TheAlgorithms/Javascript.

[45] https://tc39.es/ecma262/.

[46] "Open Enclave SDK," https://openenclave.io/sdk/.

[47] "Asylo: An open and flexible framework for enclave applications," https://asylo.dev/.

[48] J. Aumasson and L. Merino, "Sgx secure enclaves in practice: security and crypto review," *Black Hat*, vol. 2016, p. 10, 2016.

[49] "Rust SGX SDK," https://github.com/apache/incubator-teaclave-sgx-sdk.

[50] "A constant throughput, correct latency recording variant of wrk," https://github.com/giltene/wrk2.

[51] "Snabb: Simple and fast packet networking," https://github.com/snabbco/snabb.

[52] "tcpdump," https://www.tcpdump.org/manpages/tcpdump.1.html.

[53] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[54] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[55] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure multiparty computation from sgx," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 477–497.

[56] 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*. ACM, 2018, p. 2.

[57] K. A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, and R. Ankele, "Exploring the use of intel sgx for secure many-party applications," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*. ACM, 2016, p. 5.

[58] C. Shepherd, R. N. Akram, and K. Markantonakis, "Establishing mutually trusted channels for remote sensing devices with trusted execution environments," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2017, p. 7.

[59] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2435–2450.

[60] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. ACM New York, NY, USA, 2013.

[61] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel® software guard extensions: Epid provisioning and attestation services," *White Paper*, vol. 1, pp. 1–10, 2016.

[62] H. Vill, "Sgx attestation process," 2017.

[63] Y. Swami, "Intel sgx remote attestation is not sufficient," *IACR*, 2017.

[64] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P.-L. Aublin, P. Cosa, C. Fetzer, P. Felber *et al.*, "Endbox: Scalable middlebox functions using client-side trusted execution," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 386–397.

[65] S. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing security and privacy of tor's ecosystem by using trusted execution environments," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 145–161.

[66] J. Lind, I. Eyal, P. Pietzuch, and E. G. Sirer, "Teechan: Payment channels using trusted execution environments," *arXiv preprint arXiv:1612.07766*, 2016.

[67] H. Krawczyk, "Sigma: The 'sign-and-mac'approach to authenticated diffie-hellman and its use in the ike protocols," in *Annual International Cryptology Conference*. Springer, 2003, pp. 400–425.

[68] S. Balfe, A. D. Lakhani, and K. G. Paterson, "Trusted computing: Providing security for peer-to-peer networks," in *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. IEEE, 2005, pp. 117–124.

[69] M. F. Mubarak, S. Yahya *et al.*, "Mutual attestation using tpm for trusted rfid protocol," in *2010 Second International Conference on Network Applications, Protocols and Services*. IEEE, 2010, pp. 153–158.

[70] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 38–54.

[71] 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*, 2017, pp. 205–221.

[72] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 238–253.

[73] H. Wang, E. Bauman, V. Karande, Z. Lin, Y. Cheng, and Y. Zhang, "Running language interpreters inside sgx: A lightweight, legacy-compatible script code hardening approach," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 114–121.

[74] 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*, 2017, pp. 124–133.

[75] A. Ghosn, J. R. Larus, and E. Bugnion, "Secured routines: Language-based construction of trusted execution environments," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 571–586.

[76] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves." in *NDSS*, 2017.

[77] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–32, 2018.

[78] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel sgx," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 177–196.

[79] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 361–374, 2017.

[80] M. S. Melara, M. J. Freedman, and M. Bowman, "Enclavedom: Privilege separation for large-tcb applications in trusted execution environments," *arXiv preprint arXiv:1907.13245*, 2019.

[81] N. Vasilakis, B. Karel, Y. Palkhiwala, J. Sonchack, A. DeHon, and J. M. Smith, "Ignis: scaling distribution-oblivious systems with light-touch distribution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1010–1026.