CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Integrity and confidentiality for web application code execution in untrusted clients

Promoting a trust relation in web-applications

Master's thesis in Computer Systems and Networks

Asier Rivera Fernandez

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master thesis 2017

Master Thesis Report

Integrity and confidentiality for web application code execution in untrusted clients

ASIER RIVERA FERNANDEZ

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Master Thesis Report
Integrity and confidentiality for web application code execution in untrusted clients
Asier Rivera Fernandez

Supervisor: Frank Piessens, Computer Science department at the KU Leuven
Supervisor: Neline van Ginkel, Computer Science department at the KU Leuven
Supervisor: Steven Van Acker, Computer Science and Engineering department at Chalmers University of Technology
Supervisor: Magnus Almgren, Computer Science and Engineering department at Chalmers University of Technology
Examiner: Tomas Olovsson, Computer Science and Engineering department at Chalmers University of Technology

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Master Thesis Report
Integrity and confidentiality for web application code execution in untrusted clients
ASIER RIVERA FERNANDEZ
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The world-wide used web application services are crucial in today's life style and economics. However, the lack of data and execution monitoring features in web applications lead to a point in which the server can no longer trust the executions done within the client-side device. To avoid risks, developers limit the execution in the client-side devices which increases the work done by the servers. In order to promote a trust relation, we propose a solution based on Intel's SGX technology that would allow the server to delegate the execution of web application functions in the client-side device with strong security guarantees.

In order to do so, we developed a prototype called SecureJS that, first, is able to interact with the web-page submitted by the server to make the delegated code reach the native application that can run a SGX enclave, and second, is able to run the delegated code within the enclave, which offers a secure and isolated execution environment. In addition, the solution also provides remote attestation for both the correctness of the code execution and the input and output data.

The results show that the prototype increases the execution time compared to the actual state of art in JavaScript code execution, Google's V8 engine. On the other hand, the memory usage is reduced in the server side compared to the usage of NodeJS and the delegated execution to the client-side device results in reasonable memory consumption.

In conclusion, SecureJS can trigger a new area of possibilities within web application services by increasing the security guarantees and balancing the actual workload state.

Keywords: Secure javascript, web application security, Intel SGX, enclave, Chrome, SecureJS.

Acknowledgements

I would like to start expressing my sincerest gratitude to Steven Van Acker, not only for introducing me to Prof. Frank Piessens from KU Leuven, but also for being involved and supporting me along the way. In a similar way, I would like to thank Associate Professor Magnus Almgren for supporting me during the hard first part of this thesis (the administrative tasks would have beaten me without your assistance), and supervising the progress of the project. I also want to thank Associate professor Tomas Olovsson for accepting being the examiner of this thesis and having huge patience with all the obstacles faced because the thesis took place in a foreign country.

On the side of the host university, I will always be grateful to Prof. Frank Piessens for trusting me for accomplishing this thesis without knowing much about me. Last but not least, I want to thank Neline van Ginkel for all the help provided when the thesis did not seem to find the right way. It has been a pleasure to work on this project and to be part of KU Leuven.

In addition, I would like to thank my friends, the ones that I carried when I moved from Spain and the ones that made it difficult to move from Sweden. My greatest gratitude to my girlfriend for her support and always motivating me to carry on and improve and to her family for their priceless help.

Last, I would like to thank my family for their huge support. Unfortunately, they do not speak English, therefore, I appreciate your comprehension of the fact that I dedicate the following paragraph to them in Spanish.

Quiero dar las gracias a mi familia por apoyarme siempre. En especial a esas dos importantes personas sin las cuales yo no estaría aquí. Mil gracias Félix Rivera y Antonia Fernandez por vuestro apoyo, sacrificio y cariño.

Asier Rivera Fernandez, Gothenburg, July 2017

Contents

List of Figures

List of Tables

List of Listings

1
Introduction

The expansion of the web services, such as cloud services and web applications, has raised the necessity and expectations of security for client-server infrastructures. This expansion leads to functional web applications that offer a big range of services. Furthermore, these services require the usage and management of sensitive information that must be secured.

In the current web-application infrastructure, the user of a web-application trusts the web browser, the local operating system (OS), the local hardware, the client-server connection, and the web server. On the other hand, the web application provider only trusts the web server.

The reason for this lack of trust situation is that an untrusted system could interfere with the execution within the client-side device. This way, an attacker could modify or avoid the execution of Security-Sensitive Functions (SSF) and get access to the secrets included in the functions. This situation becomes more important in many web applications that offer SSF (i.e. validation and authentication) within very sensitive services, such as online banking and online health-care services. Hence, the server is forced to execute all the SSF.

Stepping away from the security concerns, and moving into performance concerns, it is also true that the situation could be balanced. All companies, whether they possess web servers that receive millions of requests everyday or only a few, are forced to execute the majority of the code within their own devices in order to guarantee the security of the data, which leads the servers to overwork in order to provide web-application services. In a case in which the servers could securely delegate part of the work to the client-side devices, those servers would benefit from a big reduction of their workload and, therefore, balance the situation. This case may be of interest for the companies in order to reduce the costs of the web-application services.

Gartner indicates that, the information security infrastructure does not support the fast changes and adaptation required by the fast-changing threat environment [14]. Therefore, vendors are shifting the security focus from individual hardware elements into software-based security. Software-based security offers a more adaptable solution and has raised the expectations on this security related area due to its flexibility. Hence, software-based security can provide solutions to web application vulnerabilities and limitations.

Based on the potential of software-based security and the necessity to give a solution to the distrust relationship, this project aims to provide an environment of trust for the server in the actual web-application infrastructure.

1.1 Goal of the project

This project focuses on a solution that would allow the server to delegate the execution of the SSF to the client-side devices. To achieve that, a secure process and environment that ensures integrity and confidentiality for the execution of the code and the results in the untrusted client-side device must be provided.

With these properties into consideration, the proposed solution, named SecureJS, is based on the new Intel SGX technology's enclaves [40] combined with the most used web browser [3], Google Chrome. SecureJS must be able to interact within Chrome with the code sent by the server, redirect the code towards the application running a secure SGX enclave, and execute the code. Furthermore, the solution must provide security guarantees through all the process without increasing the attack surface.

1.2 Attacker model

For this project we consider one of the worst case scenarios: an adversary that has control over almost the complete device. Our adversary has been able to control every hardware component, such as I/O devices, memory. However, the attacker has no control over the CPU. Therefore, the adversary is able to read and modify any memory address, alter the input values introduced by the user, etc. In addition, our adversary has also gained control over all the software implementations installed inside the device, such as the BIOS, the Operating System, and any piece of application logic on top of the Operating System. Meaning that the attacker is able to execute sophisticated attacks, such as *system call attacks* (a.k.a. Iago attacks [25]), and has access to any resource and execution carried out within the compromised device.

1.3 Relevant work

The topic of securing executions within untrusted devices has been studied in various research with different approaches. Some of those research are briefly introduced in the following paragraphs.

One of them is the solution called Haven, proposed by A. Baumann et al. in [22]. Haven focuses on offering a secure usage of the cloud services in situations where the users do not trust the service provider's remote infrastructure. In order to do so, A. Baumann et al. use the same technology that is used for this project; Intel's SGX.

Focused on a local environment, Yanlin Li et al. try to give solution to another problem of untrusted devices by introducing MiniBox [39]. In this case, MiniBox focuses on offering a two-way sandbox protection for code execution where both, the OS and the applications, get security guarantees for code execution when one or both are untrusted.

Last, Noorman et al. propose another solution called Sancus in [42]. In this case, Sancus proposes a solution for malware attacks in networked embedded devices.

Briefly, Sancus provides remote attestation for the execution of application code in the networked embedded devices, and message authentication for the communication with the networked embedded devices.

More detailed information about related works can be found in Chapter 3.

1.4 Contributions

This report is of value for the research in web application security and provides the following contributions:

- The description of the problem in web applications.
- The explanation of how Intel's SGX technology and Chrome extensions support SecureJS.
- The research investigation into a proper design of the system.
- The presentation of a solution that will help reduce the security problem.
- The performance evaluation of SecureJS.
- The security evaluation of SecureJS.

1.5 Structure of the thesis

The report is structured as follows. Chapter 1 introduces the thesis and gives a brief overview of the problem and research carried out. Chapter 2 explains concepts the reader should be familiar with in order to understand the remainder of this document. Chapter 3 describes other approaches used in research with similar problem to the one studied in this project. Chapter 4 describes the design taken in the project in order to solve or reduce the problem. Chapter 5 describes the implementation of the solution in detail. Chapter 6 introduces the tests and the results of the security and performance evaluations run over the final solution. Chapter 7 discusses the possibility of extra implementations and improvements for the final solution developed in this project. Last, Chapter 8 contains the conclusion for this document.

2
Background

The aim of this chapter is to ensure that the reader holds the required knowledge to understand the remainder of this thesis. The information given in this chapter is meant as a crash course, therefore, the reader is encouraged to consult more specialized reading material for more information about the concepts. The following sections contain information that demonstrate the importance of the project, describe the goal of the project, give an overview of the technologies used during the accomplishment of this project, and determine the adversary situation that the project encounters. Those topics are ordered as follows; a description of the actual situation of the security and trust relationship in web applications, the description of the goal of this project, an explanation of Intel's SGX technology and Chrome extensions, and a definition of the threat model and assumptions taken for this project.

2.1 The reality of web applications' security

The growth of Internet users in the last years is a fact as reported by the International Telecommunication Union (ITU) in their yearly Information and Communication Technologies (ICT) publications [15]. This reality did not pass unnoticed for the companies and governments around the world that realised the opportunity that the Internet could offer in order to help their services reach those users.

This opportunity leads to an exponential increase on the development of web applications in a short period of time. However, even if the Internet is a powerful tool that can bring a lot of benefits, there are also risks if the development is not properly evaluated in security terms. The main risk of this situation comes from the high expectancy for new web applications and the short time periods allocated for this development tasks, resulting in a lack of security evaluation and, by effect, an increase in the security risks in the web applications.

Those risks are deeply researched and studied by associations, such as Open Web Application Security Project (OWASP) and WhiteHat Security, and companies, such as Veracode. These research aim to raise the awareness of the risks that actual web applications services carry because of language related issues, among others. Although these research cover the threats generated by exploitable code for web application services in both client-side and server-side devices and this project aims to solve those at the client-side device, these research offer a good overview of the statistics for the actual risk state of web applications. The following Chapter 2.2 offers a more detailed overview of the situation in the client-side devices, the ones that are in the scope of the project.

The report “State of Software Security for 2016” published by Veracode [18], shows alarming results. Around 60% of the web applications do not pass the penetration tests done by Veracode based on OWASP policy compliance. As shown in Figure 2.1, some of the industries that offer sensitive services, such as financial, government related and health care industries, involve a high probability of containing a vulnerability within the services they offer via web applications.

Another problem related to the vulnerabilities is the time that those vulnerabilities stay open, which states the time lapse until the vulnerabilities are found. The results gathered in “Web Applications Security Statistics Report for 2016”, published by WhiteHat Security [19], show that the industry that keeps their vulnerabilities open less time is the Energy related industry, with 274 days, while the one with the highest number of days is the IT industry, with 875. This means, that vulnerabilities stay open and accessible to attackers for between 1 and 3 years, meaning plenty of time for attackers to design and execute their attacks. More information of this statistics can be found in Figure 2.2.

In addition, the same publication shows statistics about the time required for fixing the vulnerabilities once they are found. The results show that, once more, the Energy industry has the shortest periods of time, with 104 days, and the IT industry the largest, requiring 248 days to fix the vulnerability. This adds between one half and a year more of time for the attacker to exploit the vulnerabilities. More information around this statistics can be found in Figure 2.3.

In conclusion, researchers have shown the reality of the security risks in web applications. These risks are spread and numerous which requires a lot of work from the security experts. However, the development of new web applications and new technologies related to them is not waiting for those security experts to finish their research before being published, which means that it is generating more work. Moreover, in some cases, such as the services involving sensitive information related to health, financial, and personal data, the vulnerabilities imply a higher risk due to the importance of the involved data.

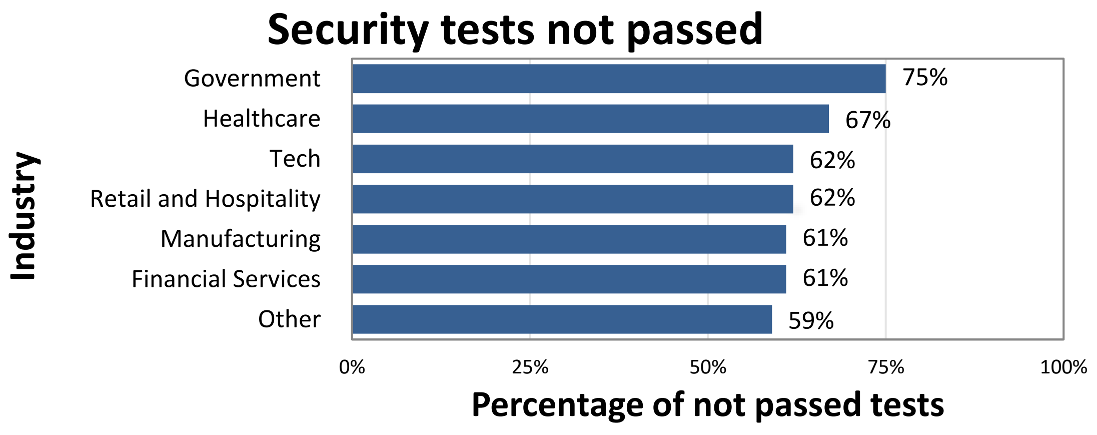


Figure 2.1: Percentage of web-applications not passing the security tests per industry. Values adapted from [18].

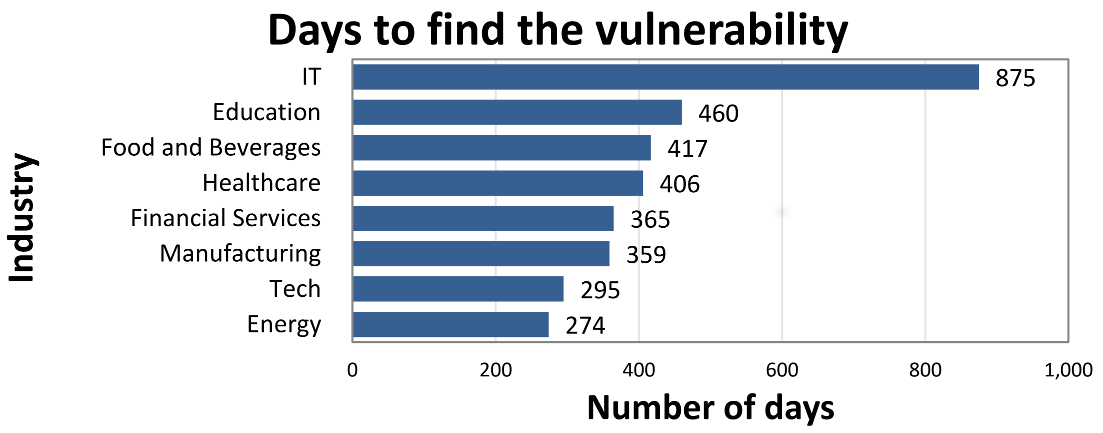


Figure 2.2: Days required to find the vulnerability per industry. Values adapted from [19].

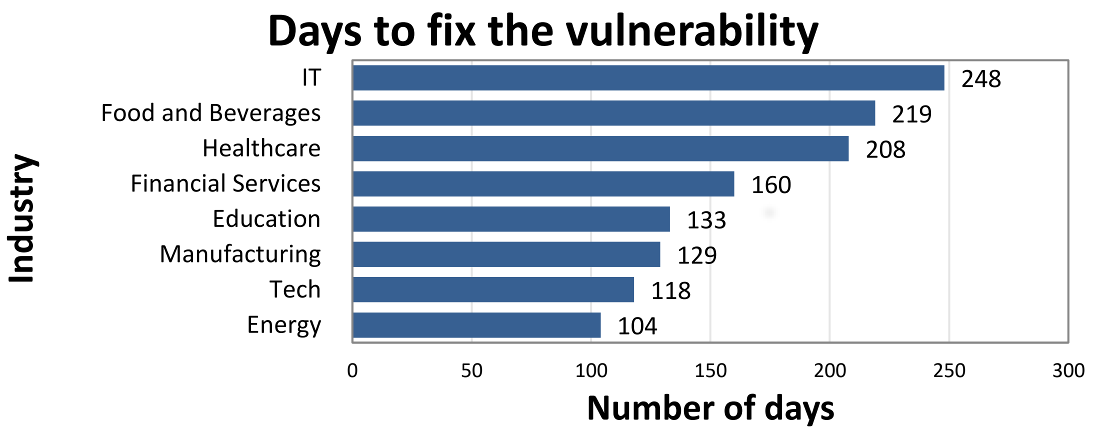


Figure 2.3: Days required to fix the vulnerability per industry. Values adapted from [19].

2.2 Trust situation in web applications

The actual trust situation between the two parties being part of the web application services, client and service provider, is unbalanced. Due to the trust relationship of both sides, where the client trusts the service provider but this does not stand in the other way, the service provider is forced to execute all the critical functions in the servers and verify every input data sent by the client.

The client trusts the service provider every time that a web-page is accessed. There exist mechanisms to ensure some guarantees for the client to trust the correctness and authenticity of the service provider, such as public key certificates [23], however, the client also adds a part of blind trust in the service provider since, as shown in Chapter 2.1, web-applications contain risks.

On the other hand, the service provider neither trusts the client-side device nor the user. This fact makes sense when reasoning about the risks that involves the delegation of execution to the client device. First, the client device state can not be verified, therefore, it can be that the device is malfunctioning or has outdated functionalities that can result in an incorrect execution of the code. Second, the client device can be compromised by an attacker or virus. The security guarantees can not be ensured by the service provider, hence, the execution of arbitrary code within the client device can be altered, which leads to a no trust situation. Third, the user can be a malicious user that is trying to alter the normal execution within its device (client device) in order to take advantage of possible vulnerabilities.

In conclusion, it is shown that the actual trust situation in web applications is affected by real problems, due to the lack of security guarantees within the client device.

2.3 Goal of the project

As has been appointed in Chapters 2.1 and 2.2, there exists a security problem that leads to a no trust between both parties in web-applications. Therefore, this project aims to solve that situation by providing security guarantees; integrity and confidentiality, for JavaScript execution in an untrusted client device.

In order to achieve that, the solution proposed in this project, named SecureJS, has to provide a secure implementation that is able to interact with web-page code, transport the data to a secure environment, set a secure environment, execute the code withing a secure environment, and return the results.

The implementation requires the usage of technologies that are further described in the following sections. Therefore, the reader is encouraged to read those sections carefully if the technologies are not familiar.

2.4 Intel SGX technology

Intel Software Guard extensions (SGX) is a new security architecture introduced for the first time in the Intel's Skylake CPU family. Intel SGX offers a new approach to the widespread situation of compromised systems and security holes by introducing a hardware assisted trusted execution environment. This new environment allows the reduction of the attacker surface to the smallest possible, the CPU boundary.

In order to better understand the further explanation of Intel SGX, we introduce the definition of some important expressions used to describe the new architecture.

Enclave. An enclave is a CPU-protected area that contains code and data within an application. This is the main expression to refer to the protected and isolated area of memory. All the data and code stored in the enclave is encrypted with a unique key for each enclave and platform and the access is restricted.

Enclave mode. Intel SGX includes a new CPU mode to the Intel CPUs. The enclave mode is a new unrestricted mode, similar to the already known privileged mode. However, the access of a user in privileged mode to the enclave mode is restricted. In addition, the enclave mode is the only one with access to the enclaves and their resources.

Untrusted Code. Part of the application code that stays out of the enclave. This code runs in a normal environment and its execution and data does not contain the security guarantees provided by Intel SGX. This part of the application code is responsible for the untrusted functions and the creation, initialization and destruction of the enclaves.

Trusted Code or Enclave Code. Part of the application code that stays inside the enclave. This code runs in an enclave environment and its execution and data does contain the security guarantees provided by Intel SGX. This part of the application code is responsible for executing the trusted functions, which deal with the security sensitive data.

Untrusted function. An untrusted function belongs to the untrusted code and it is a function whose execution takes place outside the enclave. In other words, a function executed in a normal environment without Intel SGX.

Trusted function. A trusted function belongs to the trusted code and it is a function whose execution takes place inside the enclave.

ECALL. Enclave call. A bridge function between the untrusted and the trusted code. ECALLs are divided into two types; public and private. The public ones can be called by any untrusted code while the private ones can only be called by the allowed untrusted functions defined in the configuration of the enclave.

OCALL. Outside call. A bridge function between the trusted and untrusted code. When the code within the enclave calls an OCALL, the process exits the enclave and returns the execution control to the normal environment.

Intel SGX provides the developers with a new secure environment for the execution of the applications. However, this also requires extra actions and considerations from the developers side. On the one hand, the developer must analyze the application and determine the parts of code and the data that are required to be executed within the enclave. On the other hand, the developer has to limit the usage of the enclaves to an optimized balance due to the memory and resource limitations of the enclaves. In addition, the developer must implement the code required for a proper enclave creation, usage, and destruction, including all the enclave in and out interactions done via the ECALLs and OCALLs.

A simple application runtime execution flow using an Intel SGX enclave is shown in Figure 2.4. The execution of the application will reach the part where the untrusted part creates the enclave and calls the trusted functions inside the enclave via ECALLs. Once the enclave is responsible for the execution runtime, the trusted functions will be executed. This process can contain a unique trusted function or a combination of different trusted functions being called one to another, and as many OCALL calls as needed to functions outside the enclave. Once the runtime execution within the enclave has finished, the execution exits the enclave and returns to the execution of the untrusted code, which can destroy the enclave or not (depending on the application needs), and continues with the normal execution of the application.

The process of enclave design, creation and destruction contains a large number of identity verifications, data verifications, functions, etc. that are not going to be explained in this document. Therefore, the reader is encouraged to consult more specialized reading material, such as in [40]. In addition, the ECALL and OCALL functions often include data being sent in and out of the enclave, which also includes different configuration possibilities for different types of variables supported by SGX.

The configuration related the enclave is defined in a special file named *Enclave*

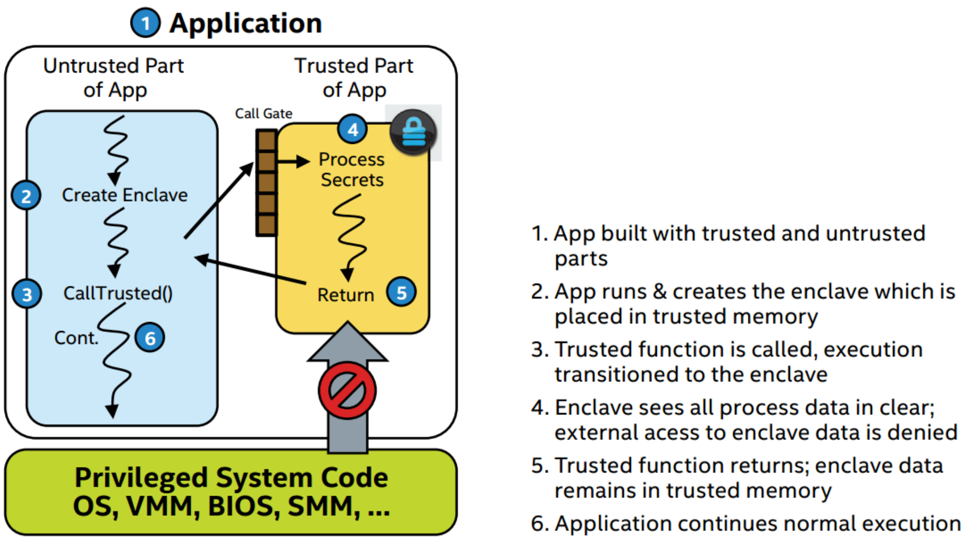

1 Application
Untrusted Part of App
Trusted Part of App
Call Gate
2 Create Enclave
3 CallTrusted()
Cont. 6
4 Process Secrets
Return 5
Privileged System Code OS, VMM, BIOS, SMM, ...
1. App built with trusted and untrusted parts
2. App runs & creates the enclave which is placed in trusted memory
3. Trusted function is called, execution transitioned to the enclave
4. Enclave sees all process data in clear; external acess to enclave data is denied
5. Trusted function returns; enclave data remains in trusted memory
6. Application continues normal execution

Figure 2.4: Runtime execution example with Intel SGX enclave. Image adapted from [17].

Definition Language (EDL). The EDL file contains a declaration of all the ECALL and OCALL functions available for the enclave as well as the properties of the data to be sent in and out of the enclave. The properties of the data cover the data type, the size of the data and the direction of the data (in or out of the enclave).

Using the new application-layer trusted execution environment, developers can ensure various security properties for commonly used applications, such as secure browsing, harden endpoint protection and secure secret storage or data protection. However, these applications require extra functionalities rather than just executing code in a secure way. Those functionalities can be local data storage, remote data storage, data sharing within the application, etc. In order to achieve that, Intel SGX provides the following functionalities:

Sealing. Intel SGX allows the secure storage of enclave secrets for persistent storage. This secure storage is done via the encryption of the data inside the enclave, using a unique *Seal key* for each particular platform and enclave, before the data is written to the hard drive.

Local Attestation & Provision. Intel SGX allows an enclave to verify the identity of other enclaves within the same device, and to securely exchange keys, credentials, and other kind of sensitive data between the enclaves.

Remote Attestation & Provision. Intel SGX allows a remote party to verify the identity of the enclave and to securely exchange keys, credentials, and other sensitive data with the enclave.

This new security architecture introduced by Intel provides a variety of security guarantees. First, SGX provides confidentiality and integrity for the application execution and data within the enclave, even under the presence of privileged malware at the OS, BIOS, VMM (Virtual Machine Monitor also known as hypervisor), or SMM (System Management Mode) layers. SGX protects the secrets even when the attacker has full control over the platform. Second, SGX offers prevention against memory bus snooping, [38] and [41], memory tampering [44], and “cold boot" attacks [33] against memory images in the RAM. And third, SGX also provides hardware based attestation capabilities to measure and verify the data and the code inside the enclaves and also the platform used by the enclaves.

2.4.1 Remote attestation

The functionality of remote attestation requires further description since it is a key component in this project. Attestation is the process of proving that an application has been properly set and initialized on a device. In this specific situation, the Intel SGX remote attestation allows a remote agent (the server in this project) to gain confidence that the intended software is securely running within a platform.

Intel SGX has designed an algorithm named Enhanced Privacy Identifier (EPID), based on the cryptographic primitive Direct Anonymous Attestation (DAA), that can accomplish a remote attestation process while ensuring the client's privacy. This algorithm makes use of unique identifiers within the platform and the server to authenticate both parties. In addition, the EPID algorithm accomplishes an Elliptic Curve Diffie–Hellman secret key generation during the remote attestation process, so that it provides a secure communication channel. More detailed information about the remote attestation process is provided in the Appendix A.1.

After a successful remote attestation process, the client and the service provider have been authenticated to each other and they share a symmetric key that can be used for data provisioning. This secure communication channel is important for this project as shown in the Chapter 5.

2.5 Chrome extensions

Chrome extensions are pieces of application logic, or small software programs, that are developed to add or enhance functionalities from the web-browser. These functionalities can be of a wide variety from the type that interferes with the actual web-page the user is viewing, such as an advertisement blocking extension, to those that give the user information about properties that are not related to the actual web-page, such as an email notifier.

All extensions share the same programming languages, which also are the most common used in the client-side browsers. Those programming languages are JavaScript, for the functionality of the extension, HTML, for the structure of the visual aspects of the extension, and CSS, for the graphical design of the extension. In addition, the extensions contain configuration storage files, such as JSON files, and can also contain: databases for local storage, functionalities for remote storage, and resource files, such as image files.

The architecture of an extension is divided into three parts, as explained in the Google's developers guide web-page [2]. First, the background page runs in an invisible manner (as the name suggests, in the background) and manages the execution flow and the behavior of the extension. Second, the User Interface pages (UI pages) contain the code required for the structure and design of the extension. Third, the content scripts manage the interaction between the extension and the web-page. And fourth, the manifest file contains the configuration of the extension.

The Background page. This page is usually defined as a JavaScript file, such as *background.js*, or as an HTML file, such as *background.html*. The HTML file normally includes JavaScript code within the HTML file or imported from an JavaScript file. The background page can be of two types: persistent or event page. The persistent page, as the name suggests, are always running while the web-browser is running. On the other hand, event pages are only invoked by the web-browser when required by the call of an event. Therefore, it is important to consider when the extension requires a page to be always loaded or not, in order to control the resource consumption of the extension.

UI pages. This kind of pages contain ordinary HTML code that displays the extension's UI. For example, they are responsible for the *options page*, where the user can change the behavior of the extension, and the *popup page* generated when the user clicks on the extension icon, where the user can interact with the functionality of the extension via buttons, input areas, etc. In addition, all the HTML files belonging to the same extension have access to each other's DOM (Document Object Model) and, therefore, they can invoke functions from each other, including the functions in the background page.

Content scripts. Many extensions' functionalities may require some kind of interaction with the web-page loaded in the web-browser and in order to achieve that, those extensions require content scripts. These scripts contain JavaScript code that is executed in the context of the web-page and not in the context of the extension. Therefore, the content scripts can read details from the web-pages loaded in the web-browser, and also modify the code of the web-page. However, the content scripts are not completely isolated from the extension and they are capable of interacting with the parent extension (in both directions) via a messaging functionality offered by the *Chrome-only APIs* (often called chrome.* APIs).

Manifest file. This file contains the configuration of the extension structured with a JSON file format. Properties such as the permission, trusted servers, name, version, and script files of the extension are defined in this file.

Although the Chrome extensions run in the web-browser's environment and can only access the resources belonging to that environment, the extensions have the possibility to interact with other native applications within the device. This functionality offers the extensions a strong tool that allows starting new instances of native applications and communicate with them. This functionality is achieved via *Native Messaging* [1] included in the *Chrome-only APIs*. In order to achieve this interaction schema, the *Native Messaging* functionality requires extra configuration for the extension, the web-browser and the native application, as will be defined later in the Chapter 5.

2.6 Threat model and assumptions

Every time a developer designs code that is going to be executed within an external device whose security guarantees are under the shadow of doubt, the trust relationship is affected to its limit. As a result, the external device becomes an untrusted device. This project focuses on returning that trust over untrusted systems by ensuring strong integrity and confidentiality guarantees for the specific no trust

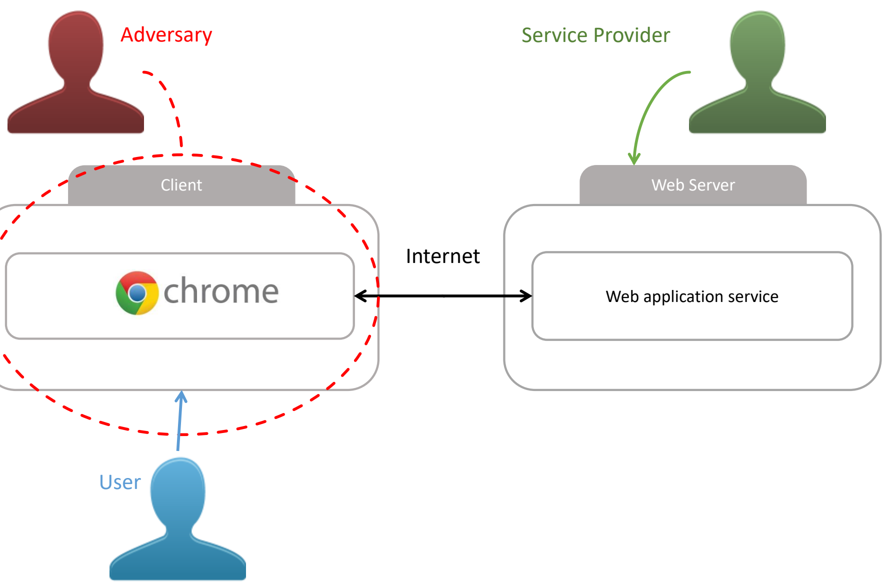


Figure 2.5: Representation of the roles in the threat model.

situation present in web applications, in the area of JavaScript to be more precise. Therefore, it is mandatory for this project to face the worst adversary possible, so that a complete trust guarantee can be offered even under the worst case scenario. A graphical representation of the roles in this situation is shown in Figure 2.5.

For this project, the adversary will affect almost the completeness of the untrusted device. We assume that the adversary has been able to gain access and control over the target device to a level in which the only part of the device out of the reach of the adversary is the Central Processing Unit (CPU). Therefore, the adversary is able to read and edit any address related to any other piece of hardware, such as ethernet controller, memory, I/O device, hard drive, and WiFi controller. This means that the adversary has control over every piece of information within the device and the information coming in and out of the device.

Additionally, the adversary has also reached to control every piece of software stored within that hardware. Critical parts of the device, the BIOS and the Operating System (OS), have fallen into the control of the adversary. The same happens with any piece of application logic running on top of the OS. Due to the control level gained by the adversary, it is possible to alter the behavior of the OS via sophisticated *System Call* attacks (IAGO attacks [25]), even when the CPU is still trustworthy.

In addition to the adversary, there are another two roles in the web application situation that the project is built around, the web application user (referred to as *user* in the future) and the web application server (referred to as *server* in the future).

In the case of the first role, the user, the situation in which the user is meant to be malicious by exploiting code injection vulnerabilities in web applications has already been well researched, i.e. in [34] and in [45]. Moreover, since the adversary is already able to modify any friendly user input and any memory address for malicious purposes, assuming a malicious user would not increase the attack surface of the actual case scenario. In addition, we also assume that the situation of users' information leakage and loss is not considered, since the device, and hence, the information within it, has already been exposed to the adversary.

In relation to the server, we enforce the trust relation by assuming a friendly behavior of the service provider. This includes a friendly behavior on the code sent by the server, but also a friendly behavior in the work balance for code execution. This means that the server will not try to reduce workload in the server by overloading the client device's work stack. The situation of the work balance is further discussed in the Chapter 7, later in the document.

In addition to the assumptions on the roles that take part in this situation, it is also necessary to explain other facts that have influence. First, the control level that the adversary has in this adversary model makes it impossible to avoid situations in which the adversary interrupts or modifies the required communications within SecureJS, with the objective of service denial. Therefore, those situations in which, SecureJS does not offer the possibility to execute the provided JavaScript code, due to the modifications done by the adversary (e.g. signature, code, encryption modifications) are defined as good, as long as, SecureJS is able to react without revealing any secret.

In fact, the aim of the project is to provide integrity and confidentiality to the JavaScript code execution, and not JavaScript execution under any circumstances. Therefore, the result of SecureJS being an error notification due to the incorrectness of the given values, is defined as a good service.

Last, the input and output data of the user is not secured, neither trusted. There exist some research around I/O protection, such as [46]. However, this project is not focused on that aspect. This decision was made based on two reasons. The user's data is already compromised by the adversary, which means that there is no point in providing security guarantees for integrity and confidentiality. Moreover, the data can not be trusted since the adversary has the access to modify it into malicious values. However, the offered solution does not check the user input via sanitization nor validation functionalities.

The implementation of those functionalities is delegated to the web application developer for two reasons; complexity and security. The web application developer has the knowledge of what inputs are permitted for the executed code, therefore, it is easier to delegate the development of those functionalities rather than adding all the possible variations of those functionalities to SecureJS. For the same reason, the fact that the web application developer has flexibility to design the required code, can add more security to the functionalities since the design can be situation dependent.

3 Related Work

The following sections contain information about related work. Some of those works and the project described in this report (SecureJS) share some similarities in the problem and the approach taken for the accomplishment of the projects: Haven, Sancus, MiniBox, and TrustJS. The first four sections in this chapter contain the description of those related works that share the most relevant similarities with *this project*, while those works that share less similar properties with *this project* are summarized in Chapter 3.5.

3.1 Haven

Researchers such as Baumann et al. have already worked around the problem of not trusting the other side's infrastructure for web services. In this case, they focused on Cloud Services and they propose a solution called Haven [22] in order to provide security guarantees for code execution within untrusted Cloud Services providers. They exposed the problem of not having full guarantees for an integral and confidential usage of the Cloud Services, such as information leakage and execution changes.

In their research, Baumann et al. assume the provider has taken the required security implementations in order to protect the provider's infrastructure against untrusted applications. However, Haven also provides protection in this area via the Drawbridge LibOS (a version of Windows 8 rebuilt to be executed as libraries in picoprocesses) and picoprocesses [11]. Together, the picoprocesses and LibOS enable sandboxing for untrusted applications with similar security to virtual machines. Hence, this combination provides protection to the host (i.e., the cloud provider) from a potentially-malicious guest.

For the implementation of Haven, Baumann et al. propose the usage of a technology developed by Intel called Software Guard extensions (SGX) [40]. SGX offers confidentiality and integrity for code execution even in the presence of privileged malware via the usage of enclaves.

This project aims to solve a situation that shares some similarities with Haven's research, although the approach is taken from the opposite point of view. Haven focuses on a situation where the client does not trust the server. On the other hand, this project focuses on the case where the server does not trust the client.

3.2 MiniBox

Li et al. identify the necessity of protecting not only the application from untrusted OS but, also the other way around, protecting the OS from untrusted applications. In order to try reduce the problem, Li et al. propose MiniBox [39], a two-way sandboxing solution that provides a two-way trust situation between the application and the OS. This project is focused on trust within one device and, therefore, it does not support any specific focus on web services or any other kind of network related applications.

To achieve the expected secure environment, Minibox's structure is based on a hypervisor named TrustVisor that isolates parts of application logics from the rest of the system. This guarantees efficient and trustworthy computing environment isolated from the OS environment.

In order to fulfill the necessity for securing the OS from the application, Minibox makes use of Google Native Client (NaCl) [47], which is a sandbox for x86 native code. NaCl offers sandboxing guarantees, which ensures the the absence of privileged x86 instructions within the native code. To achieve that, NaCl incorporates a validator that reliably disassembles the application logic and validates the disassembled instructions as being safe to execute.

This structure happens to be really similar to the structure proposed by Intel SGX, as a matter of a fact, Li et al. mention the possibility for improving their solution by making use of Intel SGX in the *Limitations and Future Work* section of their paper.

3.3 Sancus

One of the required property for this project is remote attestation. The solution must guarantee the server that the code is executed in an uncompromised environment and that the authentic data generated from the uncompromised execution is the one that reaches the destination server with security guarantees. A similar problem is researched by Noorman et al. [42].

In a networked embedded devices infrastructure, the threat of malware is a reality introduced by the combination of the connectivity and software extensibility properties offered by the infrastructure's nature.

Noorman et al. propose Sancus, which supports extensibility for remote software (even third-party software) installation on networked devices while ensuring security guarantees. Precisely, Sancus offers two properties; first, remote attestation to ensure that a specific software module is running uncompromised, and second, message authentication for the messages sent by the software module.

The problem described in Sancus's paper is very similar to the one faced during this project and gives a useful overview of both the problem and the approach taken for the solution. This information can be used to determine the necessity for remote attestation in this project. Note that, the property of remote attestation is offered by Intel SGX and it is one of the key properties in this project.

3.4 TrustJS

During the accomplishment of this project, another paper was published called TrustJS [31]. In this publication, Goltzsche et al. introduce a solution for the same problem as the one researched in this project. They also focused on providing the possibility to execute JavaScript code in an untrusted device under strong security guarantees.

In order to achieve their goal, they also made use of Intel's new technology, SGX, as it is done in this project. However, they selected another approach. Goltzsche et al. decided to use Mozilla's Firefox as the web-browser (instead of Chrome) to implement their solution and offered a different API for the web developers to make use of their solution and another enclave management within the solution.

The API for TrustJS is based on the usage of specific tags within the HTML code, so that their add-on is able to gather the data defined with the tag. This approach requires the add-on to read every web-page the client is accessing looking for those specific tags. On the other hand, the approach taken in this project offers the web developer two bridge functions that can be used in order to submit and receive the data from the solution. This way, the extension only reads the relevant data without checking every web-page, offering less workload to the device and more privacy to the client.

In addition, Goltzsche et al. designed a solution in which a pool of enclaves is created and initialized every time the web-browser is used. They developed their solution so that one enclave is assigned to each new tab opened in the web-browser. This design requires the creation of a number of processes for the enclaves that may not be used. This leads to a constant usage of memory in the client-side device. In this project, the enclave process is created and destroyed every time the solution functionality is required by the web-page. This may lead to a relatively slower performance in execution time, but it will ensure a more optimized resource usage by limiting the solution process to those situation in which it is really required.

In conclusion, both TrustJS and this project focus on solving the same problem. However, as described before, two different approaches have been selected for the solution.

3.5 Other related works

This section will contain brief summaries of other publications that can provide useful information for this project.

TaLoS. [21] Aublin, et. al. introduce a drop-in replacement for the existing TLS (Transport Layer Security) libraries. TaLoS moves the functionalities of the actual TLS to a secure environment inside an SGX enclave. This way, the TLS libraries run with all the security guarantees provided by Intel SGX.

The ghost in the browser analysis of web-based malware. [43] Provos et al. present the actual situation of malware on the Web and evidence of the importance of this rising threat. They carried out their research on a large quantity of the biggest web domains, in which they try to find how many of those domains contain any malware. The results are alarming, since a large number of those domains contain web-pages that were engaging in drive-by-downloads in order to send malware to the client-side device.

Iago attacks. [25] Checkoway, et. al. describe a new attack vector against the kernel. They show that it is possible to manipulate a sequence of integer return values to Linux system calls in order to alter the kernel's normal behavior. This shows that the task of protecting applications from malicious kernels is harder than it was once thought.

Ironclad App. [35] Hawblitzel, et. al. propose a tool that allows end-to-end security, plus full-system verification. Ironclad App allows the secure transmission of data to a remote machine with guarantees on the right execution of the application logic in the remote machine. Therefore, in addition to eliminating implementation vulnerabilities, such as buffer overflows, parsing errors, or data leaks, Ironclad App also monitors the behavior of the application at all times.

Virtual Ghost. [27] Criswell, et. al. developed a new system, called Virtual Ghost, that aims to protect applications from compromised, and even hostile, OS. This system makes use of compiler instrumentation and run-time checks on the OS in order to create memory areas (ghost memory). This ghost memory is out of the reach of the OS, which can not read or write that memory.

SecureME. [26] The fact that computing systems are becoming more distributed via mobile phones, distributed systems and cloud computing, increases the risk of an adversary obtaining physical access to those systems through theft. Attempting to solve that, Chabra et al. suggest SecureMe as a mechanism to defend applications from hardware attacks via memory cloaking, permission paging, and system call protection.

A hypervisor-based system for protecting software runtime memory and persistent storage. [28] Dewan, et. al. introduce an approach against sophisticated malware via hardware techniques. They protect the runtime memory of an application by using virtual machine monitor technologies to create a hypervisor. As a result, the runtime memory is hidden from the privileged OS without modifying the actual OS.

InkTag. [36] Hofmann et al. developed a hypervisor that aims to protect application logic from malicious or compromised OS. In this case, they make use of an *attribute-based* access control system that adds an extra access protection layer to the application resources. This way, the access control is decentralized from the OS and the application can create access control policies in order to avoid a malicious OS to access the application resources.

To sum up, as shown in the previous related works, some related works described in this chapter, such as *Ironclad* and *TaLoS*, refer to functionalities that could be used in some parts of the solution since they offer security guarantees for communications.

Other researchers have worked in the area of untrusted devices for similar problems but almost none of them for the specific problem of JavaScript execution within an untrusted client. The only one that focused on the same situation is *TrustJS*, which provides a solution with the same aim as this project. However, they used a different approach by using Firefox as the web-browser (for this project Chrome is used) and they decided to offer and API that gathers the data directly from the HTML page based on special tags created for the project while this project provides two bridge functions to interact with the HTML code.

Last, some articles refer to attacks that must be considered during the development of the solution provided in this project. Those attacks are considered to be viable for the attacker within the attacker model described in Chapter 2.6.

4 Design

This chapter discusses the general requirements of the project and the approach taken to fulfill them. The goal of this chapter is to provide the reader with an overview of the features of the solution. The following section, SecureJS: the schematic representation, provides a description of the approach taken to design complete solution. In addition, this chapter contains three more sections: Chrome extension, Host application, and SGX enclave, each of which provides detailed explanation of the functionalities of one part of the complete solution.

4.1 SecureJS: the schematic representation

The solution has to provide all the features required to achieve the main goal of the project, provide integrity and confidentiality for web application code execution in untrusted devices. In order to achieve that, the designed solution has to include the following features with strong security guarantees (see Chapter 6.2); transmission of data from the web-page to the secure environment, demonstration of the correctness and authentication of the platform, management of the data sent by the web-page, execution of JavaScript code within a secure environment, and transmission of the result value to the web-page. Figure 4.1 shows the graphical representation of the solution. The solution requires a feature that can demonstrate to the service provider the correctness of the secure environment and can authenticate the parties, so that the proposed solution can be trusted. This is an important feature of the solution, since, without a trust relation, the problematic of this project could not be solved. In order to achieve this feature, the solution makes use of the Intel's remote attestation algorithm (EPID), which allows the solution to demonstrate the correctness of the platform and authenticates both parties.

In addition, during the reasoning of the solution, two situations were defined based on the confidentiality level required. The first situation focuses on those web developers that require the complete security guarantee techniques provided by the solution, integrity and confidentiality. Some situations will require the JavaScript code and the result value to be confidential. In this cases, the web developer will make use of the solution's encryption mode. In this mode, both the JavaScript code and the result value are exchanged via sending the encryption and signature of the data, so that confidentiality and integrity are supported. This mode requires the usage of a public key pair for the signing process and a private key (symmetric key) for the encryption process.

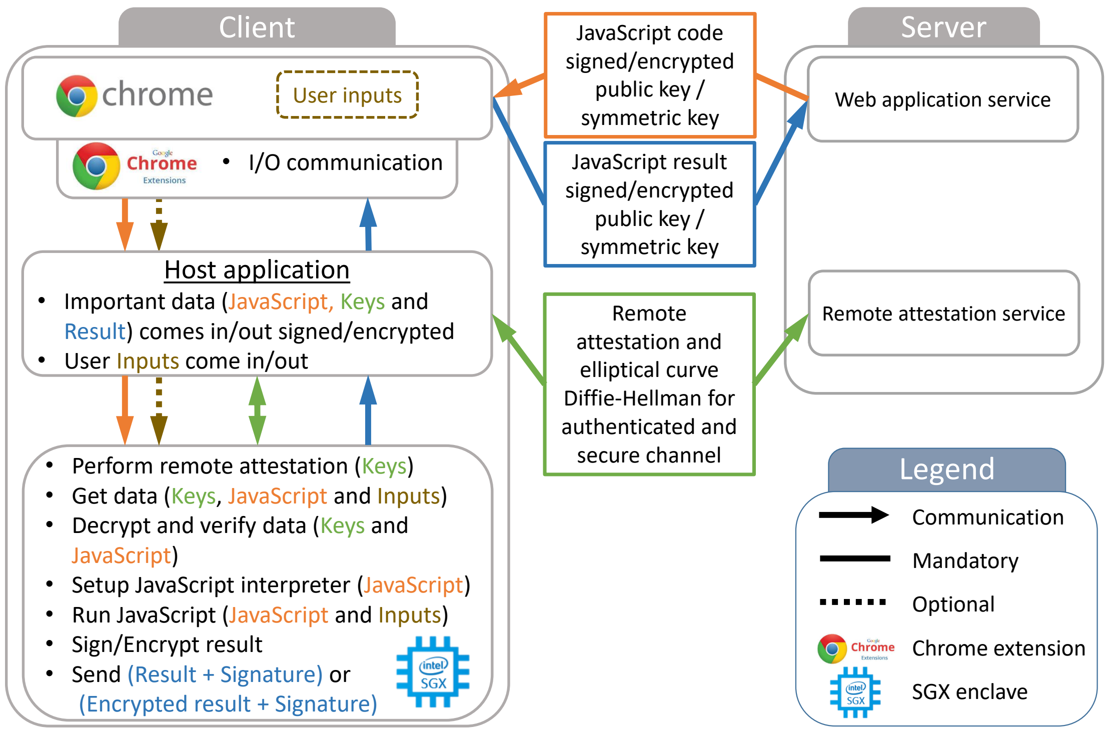

Client
chrome
User inputs
Chrome Extensions
• I/O communication
Host application
• Important data (JavaScript, Keys and Result) comes in/out signed/encrypted
• User Inputs come in/out
• Perform remote attestation (Keys)
• Get data (Keys, JavaScript and Inputs)
• Decrypt and verify data (Keys and JavaScript)
• Setup JavaScript interpreter (JavaScript)
• Run JavaScript (JavaScript and Inputs)
• Sign/Encrypt result
• Send (Result + Signature) or (Encrypted result + Signature)
intel SGX
JavaScript code signed/encrypted public key / symmetric key
JavaScript result signed/encrypted public key / symmetric key
Remote attestation and elliptical curve Diffie-Hellman for authenticated and secure channel
Server
Web application service
Remote attestation service
Legend
Communication
Mandatory
Optional
Chrome extension
SGX enclave

Figure 4.1: Schematic representation of the components and communications in SecureJS.

On the other hand, the process of encryption and decryption supposes the usage of computer execution time and resources, which may not be necessary in every case. Therefore, an extra mode has been defined, the signature mode. This mode is provided for situations in which web developers decide that it is not necessary to hide the data exchanged. In the signature mode, the JavaScript code and the result value exchange will be done via sending the plain text of the data coupled with its signature. This way, the signature mode only provides integrity to the solution and only requires the usage of a public key pair.

In order to simplify the description of the design, the solution is divided into three parts; Chrome extension, host application, and SGX enclave. Each of the sections below refers to the description of one part.

4.2 Chrome extension

The main goal of the design of the Chrome extension is to make the extension as simple as possible since it is a bridge functionality. The main goal of the extension is to provide a two way communication channel in order to exchange data between the code of the web-page and the host application. Therefore, the requirements for this part of the solution are, first, provide a two way communication link between the web-page and the extension and, second, another two way communication link between the extension and the host application.

Following the data flow in the process, the extension will provide a bridge function that the web developer's code will be able to call once all the required data has been gathered. Once the provided bridge function has been called, the extension must provide a link to the host application and send the data trough that channel. Upon the reception of the result data from the host application, the extension will be responsible for calling back the web-page in order to submit the result value.

4.3 Host application

The functionality of the host application is relatively similar to the one of the Chrome extension. In this case, the host application is required to provide a two way communication link between the Chrome extension and the host application (similar to the Chrome extension). In addition, this part has another two responsibilities; manage the creation, usage and destruction of the SGX enclave, and start and manage the connection for the remote attestation process.

For the first requisite, the host application should provide the functionality of receiving and sending data to the Chrome extension. This one is a simple functionality, this part of the host application can be described as a bridge between the Chrome extension and the enclave. This bridge functionality, coupled with the Chrome extension's bridge functionality, offers the complete bridge between the web-page and the enclave.

The second requisite forces the host application to become a more complex process. In this case, the host application will be responsible of creating, calling, listening and destroying the enclave. This requisites can easily be described by following the data flow. Given the reception of the data sent by the Chrome extension, the host application is required to, first, create the enclave environment. Second, the host application must call the enclave ECALL function related to the remote attestation. Third, it has to call the ECALL bridge function focused on the execution of the JavaScript code within the enclave and provide the data sent by the Chrome extension. Fourth, upon the reception of the output results from the JavaScript executing enclave, the host application hast to make use of the communication link to transmit the result to the Chrome extension.

The last requirement is related to the remote attestation process message exchange. Since the SGX enclaves do not support any I/O functions, the message exchange between the platform and the service provider has to be done out of the enclave. This task is delegated to the host application. Therefore, it must provide a communication service for the message exchanges.

4.4 SGX enclave

The SGX enclave is a key part of SecureJS. The usage of the SGX technology provides the required secure environment in order to offer strong security guarantees on the usage of SecureJS. In order to create a complete solution, the SGX enclave must provide the following features: cryptographic techniques, remote attestation, and JavaScript execution. These features are further described in the following sections.

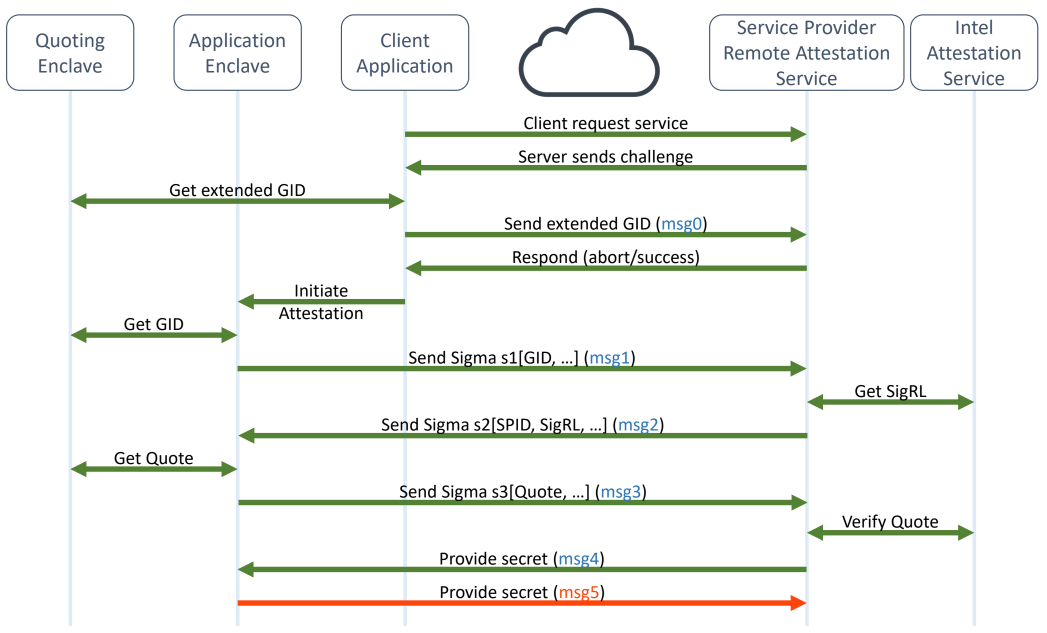


Figure 4.2: SGX infrastructure for Enhanced Privacy Identifier (EPID) algorithm with the extended functionality for SecureJS.

4.4.1 Cryptographic feature

The functionalities related to cryptography, the execution of functions, such as encryption, verification, and the resources, such as keys and hash values, are security sensitive, which makes the cryptographic techniques critical and must be secured. The design of SecureJS locates the cryptography related functionalities in the SGX enclave in order to offer security guarantees to the solution, since the SGX technology provides a secure environment out of the reach of the adversary.

As the security evaluation demonstrates (Chapter 6.2), the data exchanged in SecureJS must be protected via cryptographic techniques. Therefore, the SGX enclave must be able to hash, encrypt, decrypt, sign and verify the protected data. The protected data is composed by the JavaScript code generated by the service provider, the result value generated by SecureJS, and the data exchanged during the remote attestation process. In addition, the cryptographic techniques make use of keys that have to be generated, exchanged and protected.

4.4.2 Remote attestation feature

The objective of the remote attestation feature is, as the name indicates, to accomplish the critical task of remote attestation in order to authenticate and demonstrate the correctness of the platform to the service provider by exchanging identification values. During the remote attestation process based on the EPID algorithm designed by Intel, the service provider can send a secret in the fourth message of the protocol, see Figure 4.2.

In this project the fourth message is used for the service provider to send the keys needed to recover and verify the data sent via the web-page. However, for the correct performance of the solution, it is required that the remote attestation part carries out an extra task in addition to the one that focuses on the common remote attestation designed by Intel.

The EPID protocol message exchange is shown in Figure 4.2 and the extra task is denoted by the addition of the orange arrow that represents an extra communication message. Since the service provider is meant to decrypt and verify the result values generated from the JavaScript code executed within the JavaScript enclave, the service provider must also receive the cryptographic keys that allow those processes. Therefore, the design created by Intel for remote attestation has been extended by adding an extra step: one last message sent by the platform. Hence, the remote attestation component is responsible of implementing the necessary code to submit extra secrets to the service provider.

4.4.3 JavaScript execution feature

The main aim of the JavaScript execution feature is the execution of JavaScript code within the enclave. However, in order to achieve that goal, the enclave is required to add some extra intermediate steps related to the data gathered from the web-page. If the solution aims to offer strong security guarantees, the data sent via the web-page must be secured with cryptographic techniques as mentioned in Chapter 4.4.1. Hence, the data received from the web-page can not be directly passed to the JavaScript interpreter.

First, the JavaScript enclave has to be able to decrypt the data received. Second, it hast to verify the integrity of the data. Third, given a successful verification, the JavaScript enclave must input the data in the JavaScript interpreter. Last, it has to submit the result returned by the JavaScript interpreter to the host application. However, due to the reasons that will be shown in the security evaluation (Chapter 6.2), the result has to be secured by encryption and signature techniques.

In conclusion, the goals of the JavaScript component are summarized in, decrypt and verify the received data, set, run and listen the JavaScript interpreter, and encrypt, sign and take out from the enclave the result value.

5
Implementation

This chapter explains the implementation of the implementation of SecureJS. The aim is to describe the techniques and technologies used or developed so that the solution is able to provide the required features. The following sections provide a description of the complete SecureJS implementation and the limitations (SecureJS: the complete implementation) and the important parts that compose SecureJS: Chrome extension, Host application, SGX enclave, and Extra implementations. Each section is focused on the description of important implementation features of one part of the complete solution. Note that, the full code will not be covered in this document. Therefore, the reader is encouraged to access the available code in the git repository of the project [30].

5.1 SecureJS: the complete implementation

SecureJS is the proposed proof of concept for this thesis. SecureJS is able to interact with the web-page when required by the web-developer and offer a secure environment to execute arbitrary JavaScript code in the client-side device. A schematic representation of the implemented solution is shown in Figure 5.1. The image shows the communications and components implemented for SecureJS and the extra implementations for simulation purposes.

As a result, SecureJS fulfills the goal described in Chapter 2.3. SecureJS is composed by three components that provide different features required for the full final product. The following sections describe more in depth the implementation required for the parts that compose SecureJS: Chrome extension, Host application, and SGX enclave.

5.2 Chrome extension

The Chrome extension connects with the host application and the web-page. For the two communication requirements (web-page to Chrome Extention and Chrome extension to host application), Chrome extension's *chrome.runtime API* provides all the functions required for the message passing process [8]. First, the web-developers are provided with a function that can be used in order to initiate and send the required data to the Chrome extension. Second, the implementation also provides an asynchronous return connection, so that the extension is able to respond to the web-page once the result value of the JavaScript code is received.

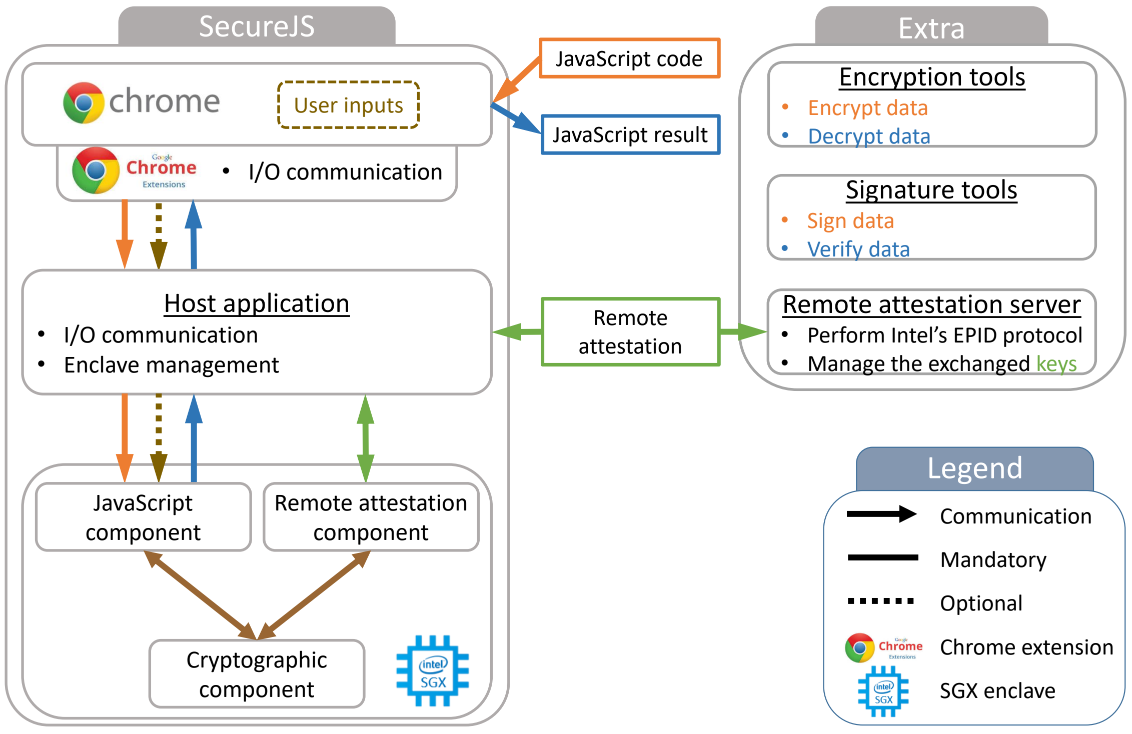


Figure 5.1: Schematic representation of the implemented components of SecureJS and the extra implementations for simulation purposes.

Additionally, the message passing feature between the web-page and the extension requires that the web-page is included in a list of allowed servers. This is done by adding the address of the trusted service providers in the configuration file of the extension (the manifest file, described in Chapter 2.5). This way, Chrome avoids a not permitted web-page from trying communicate with the extension. The solution only requires a unique extension and the allowed servers can be all added in the same manifest file. In the case that a new service provider wants to make use of SecureJS, the manifest file of the Chrome extension has to be updated in order to include the new allowed server.

On the other hand, the connection between the Chrome extension and the host application uses functions related to the Native messaging functionality also provided by *chrome.runtime API* [1]. Native messaging requires an additional configuration file (different to the manifest file of the extension) in order to know the path to the executable of the host application. This file is stored in the configuration directory of the web-browser and allows Chrome to check whether the extension is allowed to call the host application and know the path to the host application.

With the connectivity functionalities described above, all the requisites for the extension's connectivity are fulfilled and the data can be transmitted from the web-page to the host application.

5.3 Host application

The host application can be divided in two parts based on the features. The first part is related to the communication with the Chrome extension, which is a common I/O connection via the standard input and output streams. Therefore, this document does not provide a specific description of this part. The second part is related to the enclave management and services. This second part is extended below.

The enclave process management is simple: load an enclave, call the necessary ECALL and OCALL functions, and last destroy the enclave. This is the same for any program making use of the SGX technology as described in Chapter 2.

In this project, the host application initiates the remote attestation process and transmits the messages during the process. The developed code is based on the example code provided by Intel [5]. This code provides the required Untrusted and Trusted code (the trusted code is used in the remote attestation component) for a remote attestation process based on EPID. However, the example does not separate the service provider from the platform and simulates the networking parts of the remote attestation.

In order to add a real case scenario, the service provider has been completely separated from the platform and real networking functionality has been developed in between both sides. Note that, the SGX technology does not support any I/O related feature, therefore, the message exchange must be done outside the enclave. Hence, a simple socket I/O functionality has been added to the host application.

In addition, the host application is also responsible of calling the JavaScript component. Once the JavaScript component has finished, the result value generated from the JavaScript code execution is returned. This way, the host application can retrieve the result and send it via the communication channel to the Chrome extension.

With the implementation of this part, the full solution is able to receive data from the Chrome extension and transfer it to the secure environment (the enclave), start the Attestation Process, call the JavaScript component and return the result value to the Chrome extension.

5.4 SGX enclave

This section contains the description of the implementation regarding the SGX enclave. SecureJS contains three different components that run within a unique enclave: Cryptographic component, Remote attestation component, and JavaScript component. A further explanation of these components can be found in the following sections.

5.4.1 Cryptographic component

In order to simplify the tasks related to cryptography, the implementation includes an extra component, the Cryptographic component. The Cryptographic component is responsible for the storage and management of service provider and platform keys and provides the required services related to cryptography, such as data encryption and signature verification. The services of the Cryptographic component are used by both the remote attestation component and the JavaScript component.

The implementation of Cryptographic component makes use of the *Crypto* library provided by Intel. This library includes functions related to cryptographic protocols that are used for the retrieval, protection and verification of the data exchanged. In addition, the Cryptographic component also manages the keys of the full process, such as service provider's public key, symmetric key, and public key of SecureJS.

In conclusion, Crypto provides an interface for all cryptographic operations that JavaScript and remote attestation components require.

5.4.2 Remote attestation component

As mentioned before, the remote attestation component is responsible for the generation and verification and storage of the data required for the remote attestation process, such as platform correctness resources, such as QUOTE (Appendix A.1), ECDH key generation, verification of the service provider. As happens with the host application, this part also makes use of the example code provided by Intel [5] and adds some extra features to it. The features provided by Intel's example cover the normal EPID protocol process mentioned in Chapter 2 and extended in Appendix A.1. Additionally, the added features are related to the management of the secrets exchanged during the process and the extra message exchanged, as defined in Chapter 4.

The first extra feature added to the remote attestation component is related to the management of the secret sent by the service provider. The remote attestation component is responsible of decrypting and verifying the received secret. Once the secret is recovered, the secret may contain the public key of the service provider if the mode selected is Signature mode and the public key plus an extra symmetric key if the mode selected is the Encryption mode (Chapter 4.1). Those keys are then stored for common use by the Cryptographic component, also running within the enclave.

The last extra feature of the remote attestation component is to initialize the generation of the platform's secrets, such as the public and symmetric keys, that are sent to the service provider in order to be able to retrieve and verify the results of the JavaScript execution. Therefore, the remote attestation component uses the service provided by Crypto to generate and gather the required keys. Upon the creation of the required keys, the remote attestation component encrypts the keys with the ECDH key generated during the remote attestation process. The cipher containing the keys is returned to the host application and sent to the service provider as part of the extra message added to the EPID protocol for SecureJS (Chapter 4.4.2).

In conclusion, the remote attestation component consists of the already existing remote attestation example provided by Intel and provides extra features required for this specific project. Once these features are implemented, the solution is able to perform a secure remote attestation and secret provisioning process.

5.4.3 JavaScript component

As mentioned in the Chapter 4, the JavaScript component has to interpret the data related to the JavaScript code, received from the web-page via Chrome extension and host application, in order to retrieve the contents. The message is a string in JSON format. This way, the solution is able to exchange arbitrary values in a flexible manner. Therefore, a JSON based message encoding and decoding feature has been implemented within the the JavaScript component. This type of formatting uses keywords in order to structure the content of the JSON string.

Those keywords and the sample messages are described in Chapter 5.4.3.1. In order to increase the security properties of the solution, the JSON decoding feature is strict and limited to a small list of allowed keywords. Usually, JSON decoders offers a big flexibility by recognizing any keyword added to the message. However, in order to avoid giving the adversary the possibility to make use of the flexibility provided by JSON decoders, a specific and restricted JSON decoder has been implemented. The implemented decoder stops the process and sends an error message upon the reception of an unexpected value, such as a disallowed keyword, a missing mandatory keyword for the selected mode, and an allowed keyword not belonging to the selected mode.

The second feature related to the JavaScript component is the ability to retrieve and verify the received data. To achieve that, the JavaScript component uses the functions provided by Crypto to decrypt (if necessary) and verify the received data. The SecureJS process stops and returns an error message if the decryption or verification processes fail.

The last and main feature that the JavaScript component provides is the ability to execute JavaScript code within the enclave. For this purpose, the solutions require the implementation of a JavaScript interpreter. SecureJS uses MuJS, a lightweight JavaScript interpreter. MuJS is small, portable, embeddable, and secure [9] and is compatible to version ES5 of the ECMA-262 standard for JavaScript [29].

The security property is important for this project. Hence, the fact that the default setup of MuJS is sandboxed with very restricted access to resources makes it a good option as a JavaScript interpreter. Moreover, the size of the MuJS (10,000 lines of code), the portability and ability to be embedded in other applications makes it a good candidate to be integrated inside an enclave.

As mentioned before, I/O is not supported inside the enclave, which means that the code has to be checked so that it does not contain any unsupported function. Therefore, the code implemented in MuJS has been checked and the unsupported functions have been removed or modified to use supported similar functions. The main cutout from the MuJS implementation is related to time functionalities, which are not supported by SGX and have been completely removed and are not supported by the solution.

In terms of usage complexity, MuJS provides simple interface functions that easily allow the setup of an environment, the input of JavaScript code and variables, and the execution of that code.

In conclusion, the implementation of the JavaScript component allows SecureJS to retrieve the JavaScript code sent by the web-page, execute the JavaScript code in a SGX enclave, and return the result value.

5.4.3.1 JSON allowed keywords

For the message exchange features, the flexible string based JSON format has been used [6]. This format is well known for being used worldwide as an alternative to the XML format. JSON is a subset of the literal notations of JavaScript objects based on keywords to differentiate the values exchanged. This format was selected based on the possibilities that it offers to an arbitrary data exchange process. This feature is important for this project since the data exchanged can differ from one situation to other. In addition, this format is supported by the *chrome.runtime API* for the message exchanging features used for this project.

In order to allow a correct data exchange and retrieval in the solution proposed in the thesis, a number of keywords have been defined to be used for the different values exchanged through all the solution, such as CODE for the JavaScript code, MAINFUNCTION for the name of the main function in the JavaScript code, SIGNATURE for the signature of a plain text value, and ENCRYPTION for the encryption of a plain text value. Those keywords are further described in the Table A.1 in Appendix A.2. With these keywords, the web application developer can easily define and exchange data with SecureJS.

Listings 5.1 and 5.2 show an example of the JSON format message for the signature mode with and without input variables. On the other hand, listings A.1 and A.2, in Appendix A.2, show an example of the JSON format message for the encryption mode with and without input variables. As can be noticed, the construction of the messages does not require a long learning neither implementation process. However, the developer must make sure the correct keywords are used according to the mode selected.

```
{
  "CODE": "function add(var a, var b) {return a+b;}",
  "MAINFUNCTION": "add",
  "SIGNATURE": "Signature of the CODE;MAINFUNCTION in hexadecimal
      format",
  "VARIABLE": [
    {"TYPE":"int","ORDER":"0","VALUE":"1"},
    {"TYPE":"int","ORDER":"1","VALUE":"2"}
  ]
}
```

Listing 5.1: JSON format message example for signature mode with two variables

```
{
  "CODE": "function getName() {return 'Name';}",
  "MAINFUNCTION": "getName",
  "SIGNATURE": "Signature of the CODE;MAINFUNCTION in hexadecimal
      format"
}
```

Listing 5.2: JSON format message example for signature mode with no variables

MuJS requires the name of the main function to be called in order to work properly, therefore, SecureJS is required to know the name of the main function that initiates the JavaScript code execution.

On the one hand, the Signature mode fulfills this requirement by using the MAINFUNCTION JSON value. In order to provide integrity guarantees to the MAINFUNCTION the value has to be signed. That integrity guarantee is provided by the SIGNATURE parameter, which is the result of signing the JavaScript code and the name of the main function separated by a semicolon (;) character (Equation 5.1). As a result, the SIGNATURE parameter provides integrity for both the JavaScript code and the name of the main function.

On the other hand, the Encryption mode does not use the MAINFUNCTION keyword, to not provide the adversary information about the encrypted value. Therefore, the ENCRYPTION parameter must contain the encryption of the JavaScript code coupled with the name of the main function separated by a semicolon (;) character (Equation 5.2). In addition, the encryption mode also requires the usage of the SIGNATURE parameter (Equation 5.1) to provide integrity over the decrypted value from the ENCRYPTION parameter.

$$\text{SIGNATURE} = S_{(private\ key)}\{JavaScriptCode; MainFunction\} \tag{5.1}$$

$$\text{ENCRYPTION} = E_{(symmetric\ key)}\{JavaScriptCode; MainFunction\} \tag{5.2}$$

Listing 5.1 shows the usage of the JSON keyword TYPE, which is used to determine the type of the variable, inside the array VARIABLE. The variable types allowed by SecureJS and their keywords are: *str* for string type, *int* for numerical type, and *bool* for the boolean type. However, more types can be added since MuJS supports the addition of user defined variables.

Last, the three possible JSON format messages that the web-page can receive from the solution are shown in the following listings A.3, A.4, and A.5 in Appendix A.2. This way, the web application developer can easily handle the received message and act according to it, i.e. send the values to the server or start a new process upon an error message.

5.5 Extra implementations

Extra pieces of logic have been developed for testing purposes. Those applications are focused on the simulation and verification of the solution. First, a service provider's remote attestation server, that implements Intel's EPID protocol, has been implemented. Second, a signature generation and signature verification utilities have been developed. And third, encryption and decryption tools have also been created. The first program is based in the example code provided by Intel. The other two tools used for encryption and decryption purposes make use of the functionalities provided by Intel SGX for cryptography purposes, similarly to the implementation done in the Cryptographic component.

In conclusion, the extra implementations are used in order to accomplish a full simulation of the process of JavaScript execution when using SecureJS. These simulations are used for performance evaluation purposes later in Chapter 6.1.

6
Evaluation

This chapter contains the description of the tests performed and the results gathered from those tests. The tests are related to the performance, in terms of resource consumption, and to the security levels of SecureJS. The following sections are divided into performance related tests and security evaluation. Chapter 6.1 describes the resource consumption tests and presents the results. Chapter 6.2 is focused on the security evaluation accomplished over SecureJS.

6.1 Performance evaluation

SecureJS's main goal is to provide the features needed to solve the problem described in Chapter 2. However, it is also important that the usage of SecureJS is reasonable in terms of resource consumption. Therefore, SecureJS has been tested in order to check the performance.

The web-browser Chrome is able to run JavaScript code thanks to Google's JavaScript interpreter named *V8* [13]. This interpreter offers high performance in code compilation and execution and it can be embedded into other applications, which makes it the selected choice in many projects, such as Chromium, NodeJS, and other applications. In conclusion, V8 is the current "state of art" JavaScript interpreter.

NodeJS is a cross-platform JavaScript runtime environment based on the V8 interpreter that provides the possibility to execute JavaScript code in the server-side device [10]. This environment has been used in the tests in order to compare its performance to the one of the proposed solution.

As described in Chapter 3.4, during the accomplishment of the thesis David Goltzsche et al. published a paper in which they introduced TrustJS [31]. TrustJS focuses on solving the same problem as SecureJS with a different approach. Therefore, TrustJS is meant to be the direct opponent for SecureJS. As a result, the performance results presented in the TrustJS paper are used for comparison against SecureJS.

The tests developed for this project aim to gather data that allow the comparison of the performances of a web-application service between the currently used setups (JavaScript execution in Chrome and NodeJS), TrustJS, and SecureJS. In order to do so, a number of tests levels and setups have been defined as described in Chapter 6.1.1.

6.1.1 Tests definition

In order to analyze SecureJS's performance properly, five setups have been developed. Those setups provide information of different situations, which some make use of the proposed solution and others do not. This way, SecureJS can be compared against real implementations that are nowadays used worldwide. The list of tests designed is the following:

1. JavaScript execution in encryption mode.
2. JavaScript execution in signature mode.
3. JavaScript execution in NodeJS (server-side device).
4. JavaScript execution in Chrome (client-side device).
5. Extra: JavaScript execution in MuJS.

The first two setups are meant to provide performance data about SecureJS. This data can be opposed to the results gathered in the third and third test setups. Last, the MuJS setup aims to provide a comparison between the performance of the MuJS interpreter in a normal environment and in an SGX enclave. Each of the tests provide execution time results for the complete setup (named as macro-benchmarks in advance) and, in some cases, also the execution time values of internal parts are gathered (named as micro-benchmarks in advance). In addition, the test setups provide data in terms of memory consumption for the main applications used in each setup.

The macro-benchmark results are a result of generating timestamps before and after the execution of the complete setup, from the moment in which the server generates the page to answer the request of the client until the reception of the result data in the server. In addition, some components of SecureJS also include the generation of timestamps during the execution for micro-benchmark results. As a result, the micro-benchmark results show a deeper insight of specific parts of a component, such as the host application and the JavaScript component inside the SGX enclave.

The memory consumption is measured by Linux's command *time* [7]. Time command allows the monitoring and result output of a number of resources consumed by the given process, such as CPU usage, memory usage, and number of signals delivered to the process. The following paragraphs provide further description of the test setups.

6.1.1.1 Encryption mode and signature mode setups

Encryption mode and signature mode setups aim to test the encryption mode and the signature mode, respectively, of SecureJS. Both setups also include micro-benchmark results for the runtime of a number of parts within SecureJS: Chrome extension, host application, remote attestation component, and JavaScript component. Moreover, the JavaScript component also offers deeper insights of the runtime execution within the component, such as JavaScript code execution time, variable load time, data signing time, and data decryption time. In reference to the memory consumption, the setups offer results for the host application and the remote attestation service that is run by the service provider.

6.1.1.2 Server-side NodeJS setup

The NodeJS setup aims to gather information of a real situation where the code execution is delegated to the server-side device. In this setup, the JavaScript code is executed in the NodeJS environment that is requested by the web-application. In addition, no micro-benchmark results are gathered, however, the memory consumption generated by the execution of the JavaScript code inside NodeJS is measured.

6.1.1.3 Client-side Chrome setup

The Chrome setup tests a simulation accomplished in a current Chrome browser. This setup does not include the usage of SecureJS and simply executes the given JavaScript code in Chrome as any current web-page. The Chrome setup does not provide any data regarding micro-benchmark results nor the memory consumption generated for the execution of the JavaScript code. This setup focuses on offering a base line to evaluate SecureJS.

6.1.1.4 MuJS setup

The MuJS setup is an extra setup that focuses on the measurements of a MuJS implementation for JavaScript execution in a normal non-protected environment. The goal is to compare the runtime performance of MuJS inside and outside an SGX enclave. In order to achieve that, MuJS setup provides the time consumption results from the execution of JavaScript code outside of an enclave. These results can be compared to the results obtained from the Encryption Mode and Signature Mode setups: the micro-benchmark results regarding the embedded version of MuJS in the JavaScript component of SecureJS.

6.1.1.5 JavaScript code for testing

Each of the tests executes the same JavaScript code. For instance, the code selected is a simple implementation of a prime finder function that looks for all the existing prime numbers within a given range of numbers by iterating over all the possible divisors in order to determine whether a number is prime. Optimized versions of this type of function exist, such as the Sieve of Eratosthenes algorithm [32]. However, the aim is to test the performance of SecureJS and, therefore, the optimization of the code is not important.

In addition, this code allows the addition of different levels to the tests by simply increasing the range of numbers given to the function. This increase affects the iterations run by the code in an quadratic way, which allows us to test higher resource consuming executions, in terms of execution time and memory usage. This option has been used to test every setup in three different levels based on the resource consumption; low, medium and high. Each of the levels takes a different range; 1K, 10K and 100K, respectively. Last, each of test setups in each level has been run for a number of repetitions in order to gather more data; 100, 100 and 10, respectively. The reason for reducing the high level tests to 10 repetitions has been the fact that the high level execution required a long time for each repetition, see the results of related to high level tests in Chapter 6.1.2.

6.1.1.6 Web-application server setup

As mentioned in Chapter 5, SecureJS implements a real networked connection for the remote attestation process. Moreover, the web application service is deployed in an Apache server and, therefore, it is also a real networked situation. However, both client and server side services were deployed in the same device. This means that the execution times related to the communication process may be lower in this scenario rather than in a real world situation, were the server usually is far from the client and the connection may be slower. Additionally, the setups in which the data is generated in the client-side device (first three setups in the list above) send the data generated to the server. Hence, the communication times related to the web-application service are affected by the quantity of data generated.

The following sections contain the results for the tests defined above and the resultant conclusions.

6.1.2 Runtime performance

This section shows the results gathered during the accomplishment of the tests described in Chapter 6.1.1. The following sections are divided based on the benchmark level; macro and micro. Each of the sections refer to the group of figures that belong to macro or micro-benchmarks and define the data represented by each of the figures.

6.1.2.1 Macro-benchmark results

Figures 6.1, 6.2, and 6.3 represent the data for the macro-benchmark results for the low, medium, and high level tests respectively. The results represent the execution times required by each of the setups to accomplish a complete simulation of a web-page request that contains the execution of the JavaScript code defined above. These representations can be used for the performance comparison between the current web-application setup and the setup of SecureJS.

6.1.2.2 Micro-benchmark results

In order to offer a better understanding of the performance of SecureJS, micro-benchmarks have been set. The results of those tests are shown in Figures 6.4, 6.5, and 6.6 for each of the test levels; low, medium, and high respectively. These results may help determine which of the parts within SecureJS affect the performance of the complete solution, which can be useful to determine the areas that should be optimized.

Figure 6.7 shows the results gathered over the micro-benchmarks for the JavaScript component in Encryption and Signature Modes. These results give information about the time required by the tasks that JavaScript component carries out. Additionally, Figure 6.8 shows the micro-benchmark resulting from the execution of the high level JavaScript code of the Encryption Mode, Signature Mode and MuJS setups.

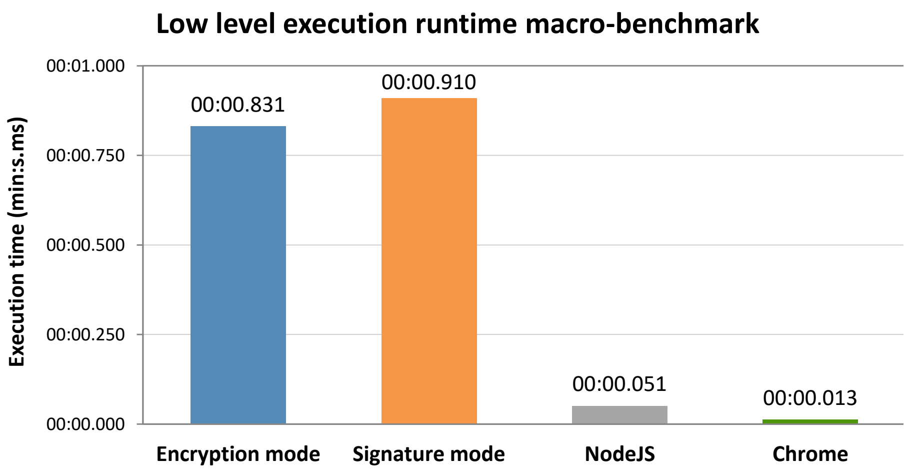


Figure 6.1: Low level execution time macro-benchmarks of each setup.

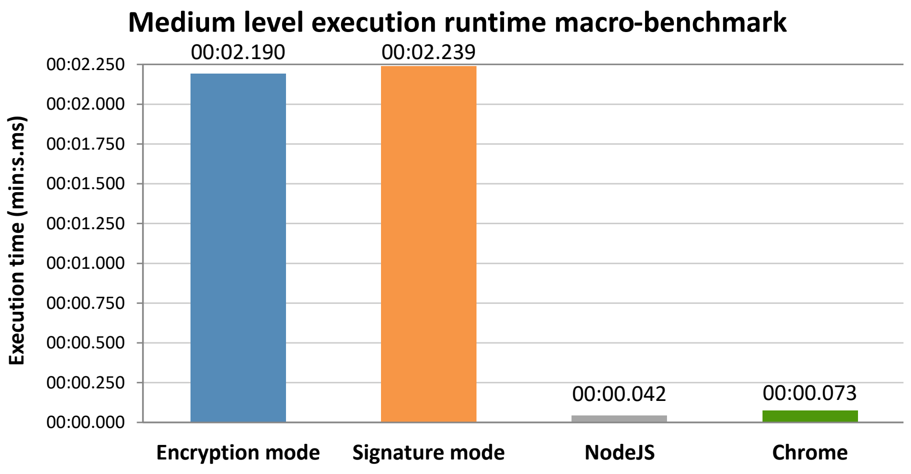


Figure 6.2: Medium level execution time macro-benchmarks of each setup.

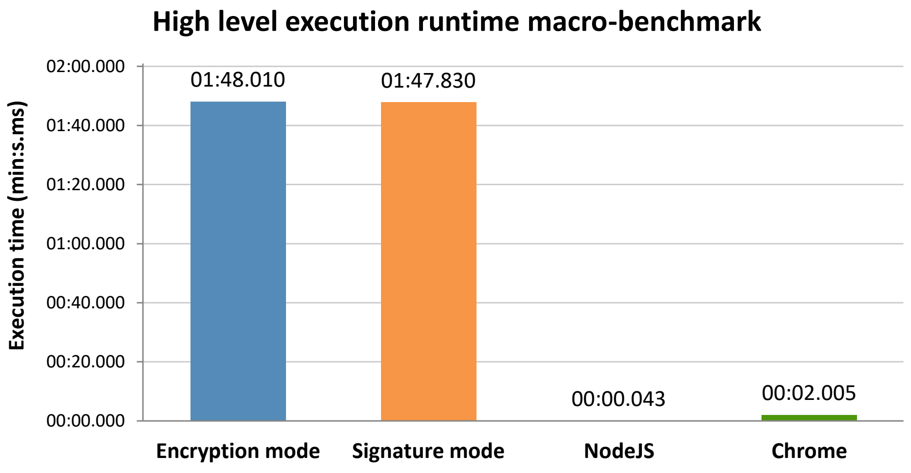
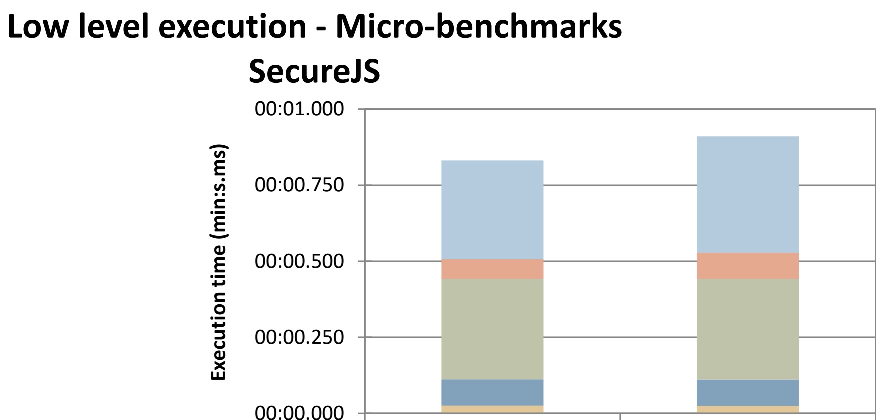
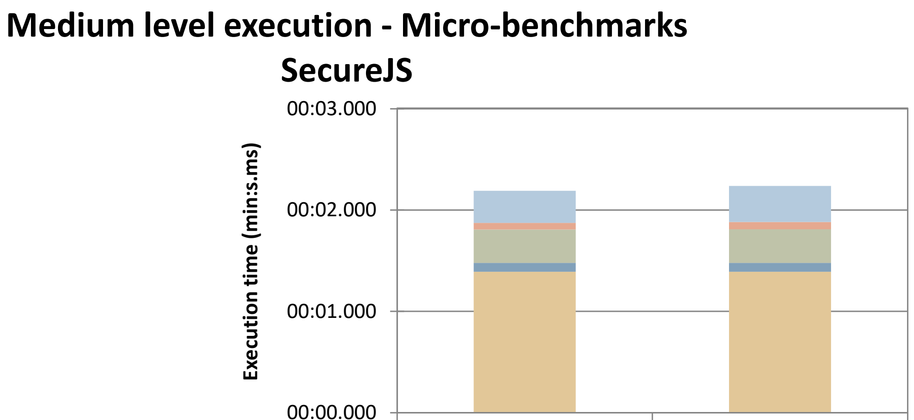


Figure 6.3: High level execution time macro-benchmarks of each setup.



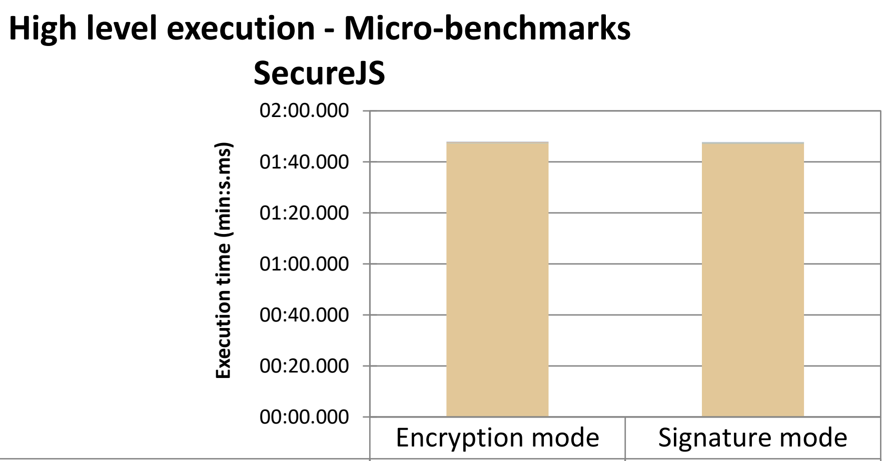
	Encryption mode	Signature mode
🟦 Server ↔ Client communication	00:00.325	00:00.383
🟧 Extension ↔ Host application communication	00:00.065	00:00.086
🟩 Enclave management	00:00.330	00:00.331
🟦 Remote attestation component	00:00.086	00:00.087
🟨 JavaScript component	00:00.025	00:00.024
Total time	00:00.831	00:00.910

Figure 6.4: Low level micro-benchmark results of SecureJS for encryption mode and signature mode setups.



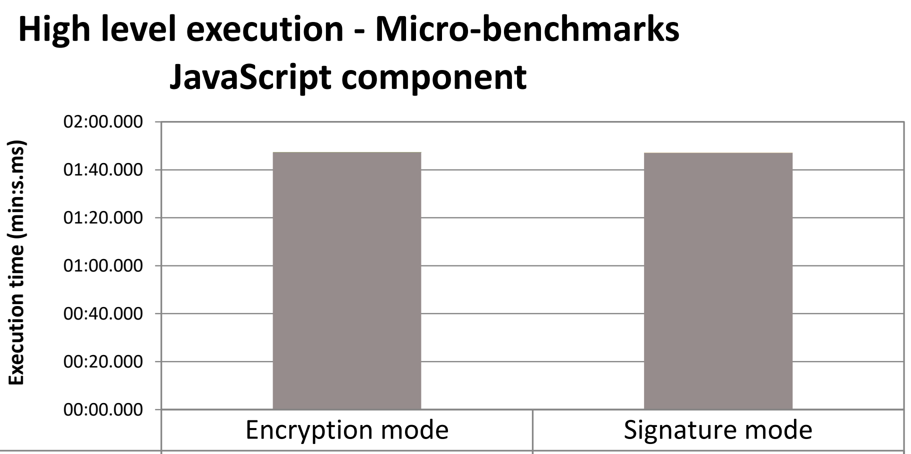
	Encryption mode	Signature mode
🟦 Server ↔ Client communication	00:00.315	00:00.355
🟧 Extension ↔ Host application communication	00:00.065	00:00.073
🟩 Enclave management	00:00.331	00:00.332
🟦 Remote attestation component	00:00.086	00:00.086
🟨 JavaScript component	00:01.392	00:01.392
Total time	00:02.190	00:02.239

Figure 6.5: Medium level micro-benchmark results of SecureJS for encryption mode and signature mode setups.



	Encryption mode	Signature mode
🟦 Server ↔ Client communication	00:00.197	00:00.282
🟧 Extension ↔ Host application communication	00:00.196	00:00.012
🟩 Enclave management	00:00.327	00:00.330
🟦 Remote attestation component	00:00.086	00:00.086
🟨 JavaScript component	01:47.204	01:47.120
Total time	01:48.010	01:47.830

Figure 6.6: High level micro-benchmark results of SecureJS for encryption mode and signature mode setups.



	Encryption mode	Signature mode
🟧 Code cipher decryption	00:00.000	
🟩 Code signature verification	00:00.000	00:00.000
🟨 Time to load the code	00:00.001	00:00.000
🟩 Time to load the variables	00:00.000	00:00.000
⬛ Time to execute the code	01:47.195	01:47.119
🟫 Result signature	00:00.001	00:00.001
🟦 Result encryption	00:00.007	
Total time	01:47.204	01:47.120

Figure 6.7: High level micro-benchmark results of the JavaScript component for encryption mode and signature mode setups.

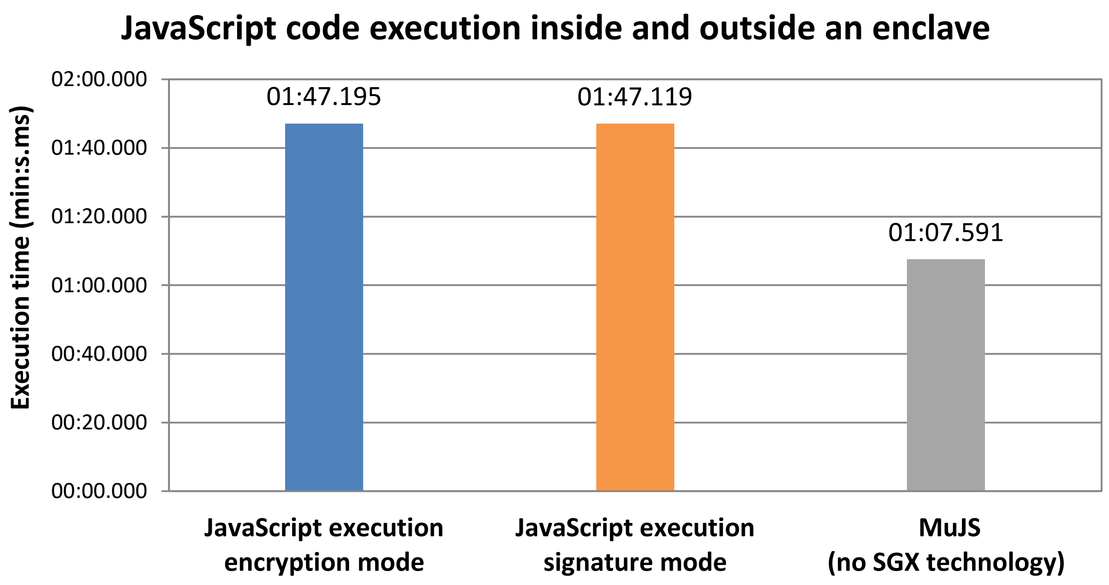


Figure 6.8: JavaScript code execution time results inside and outside an enclave.

6.1.2.3 SecureJS vs TrustJS

David Goltzsche et al. show the results of their performance evaluation in [31]. However, the test designed for TrustJS are different from the ones designed for the thesis. In the case of TrustJS, the JavaScript code selected has been an empty loop that iterates for 5000 rounds. The complexity of the code of TrustJS is lower than the code used for the low level tests run in the thesis. However, we use the results gathered from the low level tests for comparison between TrustJS and SecureJS.

Figure 6.9 shows the results taken from the paper of TrustJS [31] in comparison with the values gathered from the low level tests for SecureJS, both in Encryption and Signature Modes. These results coupled with the micro-benchmark results shown in Figure 6.4 are useful for discussing the differences in performance between both solutions. This discussion can be found later in Chapter 7.

6.1.3 Memory performance

The memory consumption values gathered during the performance of the tests are shown in Figure 6.10. That figure shows the memory consumption of the three applications (host application, remote attestation Server, and NodeJS server) used during the testing for each of the test levels described above. Note that, the consumption is related to a unique user and per execution request. In addition, Figure 6.11 shows the base memory usage of the the host application server and the NodeJS environment.

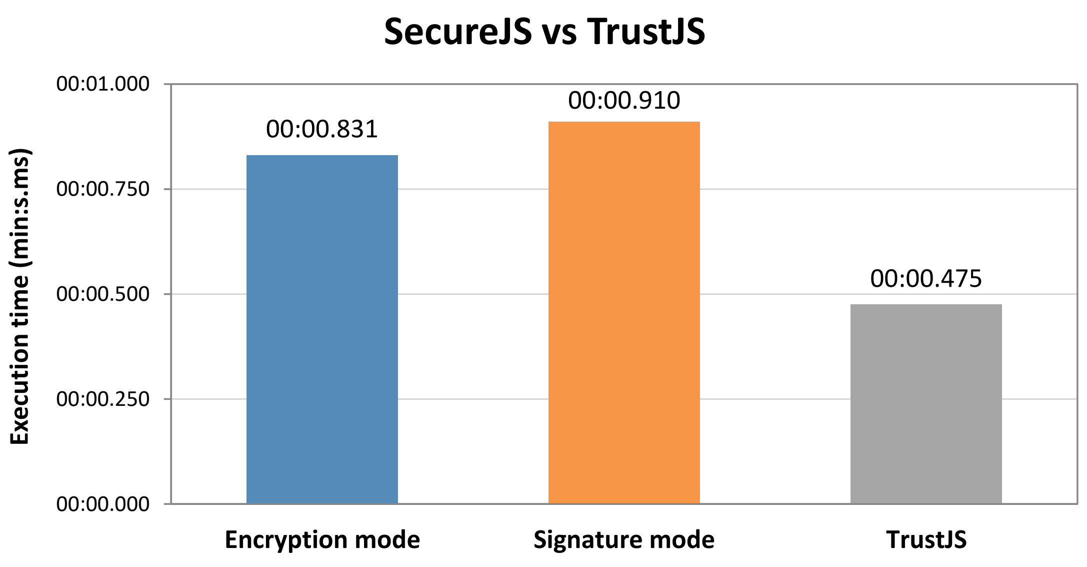


Figure 6.9: Execution times comparison between SecureJS and TrustJS.

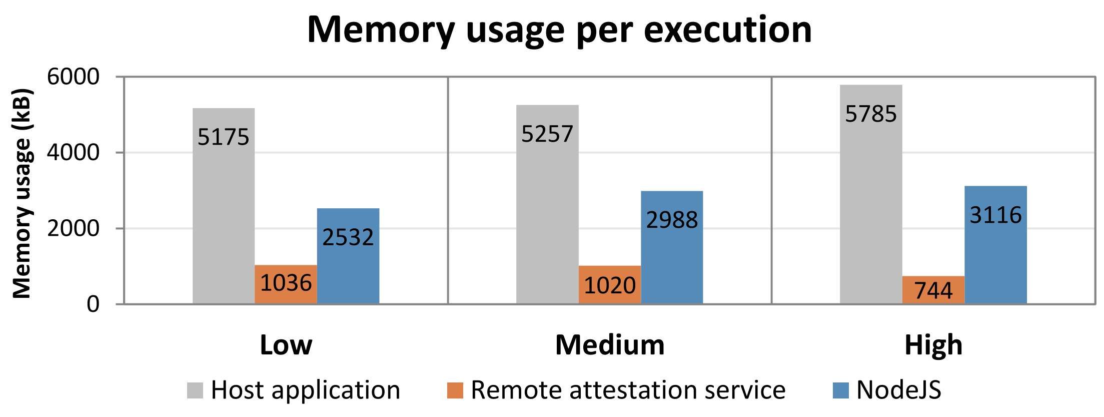


Figure 6.10: Results of the memory usage per execution.

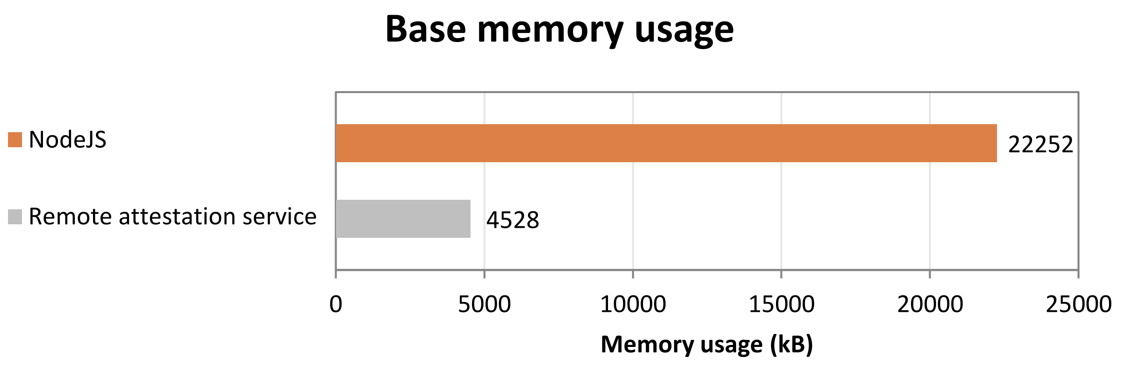


Figure 6.11: Base memory usage of remote attestation server and NodeJS environment.

6.2 Security evaluation: STRIDE

The STRIDE model was created by Microsoft as part of the process of threat modeling [12]. STRIDE helps security experts reason and find threats to a system. During the modeling process, a graphical model of the target system is used as assistance. This graphical model of the target system can be constructed in parallel to the STRIDE modeling process and includes a full breakdown of the processes, data storage technologies, data flows and trust boundaries.

STRIDE provides short and direct descriptions of security threats divided in six categories:

- **Spoofing identity.** An example of identity spoofing is illegally accessing and then using another user's authentication information, such as username and password.
- **Tampering with data.** Data tampering involves the malicious modification of data.
- **Repudiation.** Repudiation threats are associated with users who deny performing an action without other parties having any way to prove otherwise.
- **Information disclosure.** Information disclosure threats involve the exposure of information to individuals who are not supposed to have access to it.
- **Denial of service.** Denial of service (DoS) attacks deny service to valid users.
- **Elevation of privilege.** In this type of threat, an unprivileged user gains privileged access and thereby has sufficient access to compromise or destroy the entire system.

All the six categories are considered by the experts while reasoning about the possible threats. In order to ensure that no step of the modeling is missed, the STRIDE threat modeling process is divided in simple steps. This makes it easier for the experts to follow the process without missing any critical portion. The threat modeling process is structured as follows:

1. Identify the known threats to the system.
2. Rank the threats in order by decreasing risk.
3. Determine how to respond to the threats.
4. Identify techniques that mitigate the threats.
5. Choose the appropriate technologies from the identified techniques.

As can be seen in the list above, step 2 focuses on the ranking of the threats. In order to rank the threats, there exists a evaluation and calculation process to give a numerical value to each of the threats. This process is shown in the list below.

1. Evaluate the chance of a threat to happen: chance = [1,10] (high to low chance).
2. Evaluate the damage if a threat occurs: damage = [1,10] (low to high damage).
3. Calculate the risk of a threat: risk = damage / chance

The step 4 of the threat modeling process structure, requires the identification and selection of the appropriate mitigation techniques. Therefore, the experts can make use of different techniques, such as Authentication, Authorization, Hashing, and

Digital Signatures, to mitigate threats for each of the six threat categories defined in STRIDE.

6.2.1 STRIDE modeling for SecureJS

For the evaluation of SecureJS developed in this project, a STRIDE modeling was carried out. As mentioned before, this process requires of a graphical model that will help to reason and find threats in the system. The graphical model used for this project can be seen in Figure 6.12. In the figure, the red line defines the trust boundary, the data flow, data storage and processes inside the red line is defined as the untrusted area while the areas outside the red delimiter are meant to be trustworthy. Before starting reasoning and finding threats in SecureJS, it is necessary to define the assumptions that affect this project.

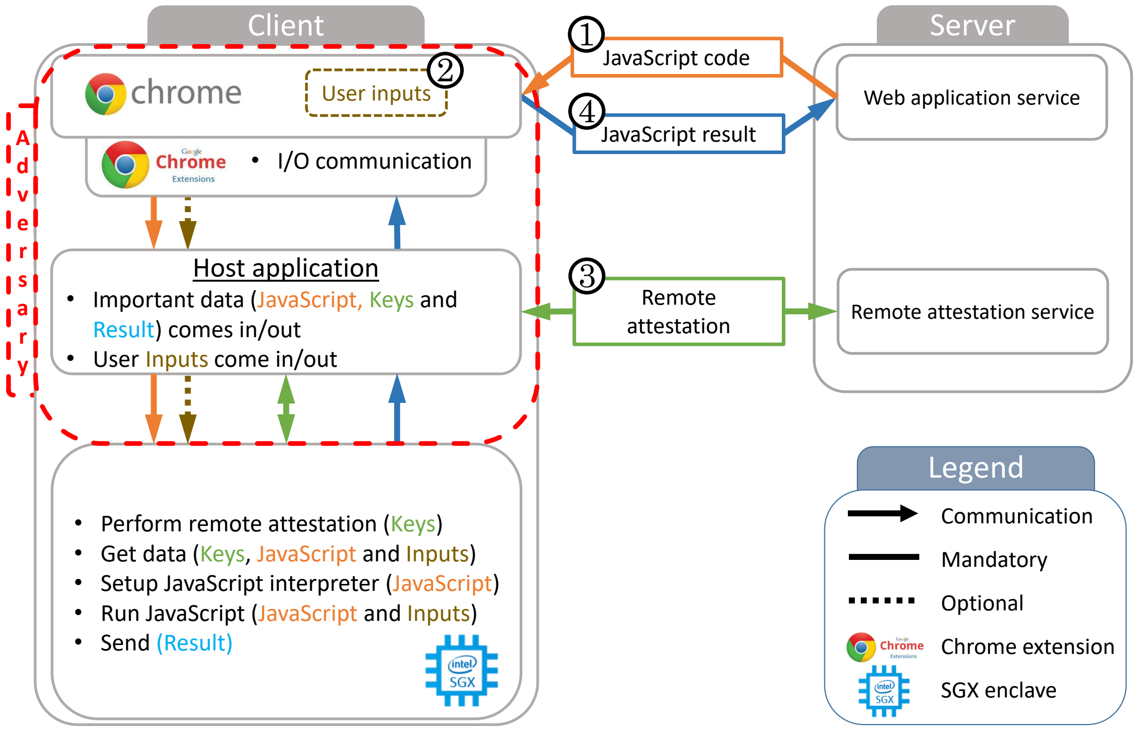


Figure 6.12: Graphical representation of the data flow in SecureJS.

First, as mentioned in Chapter 2, any Denial of Service (DoS) attack done by the adversary in the platform is negligible due to its high privilege level.

Second, the elevation of privilege threats within the client-side platform are also negligible since the adversary already has the highest privilege level in the platform (it is assumed that the enclave mode is not accessible for the adversary).

Third, the possible threats against the remote attestation service provided by the service provider are not going to be reasoned in this STRIDE modeling procedure. This decision is based on the assumption that the service provider rely on security experts that ensure security guarantees for the services provided by the service provider in order to avoid any known threat.

Fourth, the possibility of the adversary to spoof the identity of the service provider's web application services is assumed to be mitigated by the state of art of web application authentication techniques, such as certificates [23].

The following paragraphs represent each of the threats reasoned and found during the accomplishment of the STRIDE modeling process. Each paragraph will describe the threat, show the ranking evaluation reasoning, and define the security techniques used to avoid the threat.

6.2.1.1 Threat Nº 1

Definition. The adversary can tamper with the JavaScript code sent by the service provider (in the Figure 6.12 denoted as ①) once the code is in the client-side device and before it reaches the enclave.
Risk 10. The chance for this threat to happen is 1 (very high) since it would be easy and attractive for the adversary to change the code into a malicious one. The damage generated by this attack is 10 (very high) since this attack is breaking one of the main goals of SecureJS, the integral execution of a given code.
Mitigation. In order to mitigate this threat, the service provider is required to send the code and its signature (generated by hashing the code and then signing the hash value) for the Signature Mode. While in the Encryption Mode, the service provider is meant to send the encryption and GCM MAC value of the code in addition to the signature (hash and sign the code).

6.2.1.2 Threat Nº 2

Definition. The JavaScript code sent by the service provider (①) can be exposed to the adversary once the code reaches the client-side device and before it is sent to the enclave.
Risk 10. The chance for this threat to happens is 1 (very high) since it would be easy and attractive for the adversary to access the code and get knowledge about it. The damage generated by this attack is 10 (very high) since this attack is breaking one of the main goals of SecureJS, the confidential execution of a given code.
Mitigation. This threat only affects the Encryption Mode. In this mode, this threat is mitigated by using the same techniques as in Threat Nº 1. The service provider is meant to send the encryption and GCM MAC value of the code in addition to the signature (hash and sign the code).

6.2.1.3 Threat Nº 3

Definition. The secrets sent during the remotte attestation process (③) can be tampered with malicious values by the adversary.
Risk 10. The chance for this threat to happens is 1 (very high) since it would be easy and attractive for the adversary to modify the values of the messages exchanged to its own benefit. The damage generated by this attack is 10 (very high) since this attack would allow the adversary to be the MITM (Man In The Middle) and manage the keys exchanged, the result is breaking both main goals of SecureJS, the integrity and confidentiality of SecureJS.

Mitigation. In order to mitigate this threat, the the process makes use of the EPID protocol designed by Intel, which makes use of a hash, a MAC and encryption techniques to protect the secret. In addition, once transmitted, the secrets are stored in the enclave, which protects them from the adversary.

6.2.1.4 Threat Nº 4

Definition. The secrets sent during the remote attestation process (③) can be exposed to the adversary.
Risk 10. The chance for this threat to happens is 1 (very high) since it would be easy and attractive for the adversary to read the messages sent during the remote attestation process. The damage generated by this attack is 10 (very high) since this attack exposes critical secrets, such as the signature and encryption keys, for the security guarantees provided by SecureJS.
Mitigation. In order to mitigate this threat, the process makes use of the EPID protocol designed by Intel, which makes use of a hash and MAC techniques to protect the secret. In addition, once transmitted, the secrets are stored in the enclave, which protects them from the adversary.

6.2.1.5 Threat Nº 5

Definition. The result value from the JavaScript execution (④) that is sent by the platform to the service provider can be tampered with malicious values by the adversary once the value exits the enclave and before it is sent to the service provider.
Risk 10. The chance for this threat to happen is 1 (very high) since it would be easy and attractive for the adversary to change the value into values of interest. The damage generated by this attack is 10 (very high) since this attack is breaking one of the main goals of SecureJS, the integral execution of a given code.
Mitigation. In order to mitigate this threat, the process makes use of the EPID protocol designed by Intel, which makes use of a hash, a MAC and encryption techniques to protect the secret. In addition, once transmitted, the secrets are stored in the enclave, which protects them from the adversary.

6.2.1.6 Threat Nº 6

Definition. The result value from the JavaScript execution (④) that is sent by the platform to the service provider can be exposed to the adversary once the value exits the enclave and before it is sent to the service provider.
Risk 10. The chance for this threat to happens is 1 (very high) since it would be easy and attractive for the adversary to access the code and get knowledge about it. The damage generated by this attack is 10 (very high) since this attack is breaking one of the main goals of SecureJS, the confidential submission of the result value.
Mitigation. This threat only affects the the Encryption Mode. In this mode, this threat is mitigated by using the same techniques as in Threat Nº 1, the service provider is meant to send the encryption and GCM MAC value of the result value in addition to the signature (hash and sign the result value)

6.2.1.7 Threat Nº 7

Definition. The adversary can spoof the identity of the service provider's remote attestation service (③), which would allow the adversary to control the remote attestation process.
Risk 5. The chance for this threat to happens is 2 (very high) since it would be a bit more difficult for the adversary to develop the required service but still attractive to control the values of the messages exchanged to his own benefit. The damage generated by this attack is 10 (very high) since this attack would allow the adversary to be the MITM (Man In The Middle) and manage the keys exchanged, the result is breaking both of the main goals of SecureJS, the integrity and confidentiality.
Mitigation. In order to mitigate this threat, the process makes use of the EPID protocol designed by Intel, which makes use of a hash, a MAC and encryption techniques to protect the secret. In addition, once transmitted, the secrets are stored in the enclave, which protects them from the adversary.

6.2.1.8 Threat Nº 8

Definition. The user input (②) can be tampered with malicious values by the adversary in order to modify it into malicious code that would alter the JavaScript code execution.
Risk 3. The chance for this threat to happen is 3 (high) since it would be really easy for the adversary to modify the values of the user input, however, finding the necessary malicious values to achieve the goal makes it more complicated. The damage generated by this attack is 9 (very high) since this attack would allow the adversary to alter the execution and the result is breaking one of the main goals of SecureJS, the integrity.
Mitigation. In order to mitigate this threat, SecureJS requires the web application developers to build tamper-resistant functions and apply validation functionalities to the code.

6.2.1.9 Threat Nº 9

Definition. The service provider can repudiate the JavaScript related data (①) sent to the client platform.
Risk 1. The chance for this threat to happens is 7 (low) since during the accomplishment of this project it is assumed the good will of the service provider and it can be assumed that the chances for the service provider to repudiate his own data is low. The damage generated by this attack is 7 (medium) since this threat could result into a problem, for example in the case that the JavaScript code affected the state of the client platform.
Mitigation. In order to mitigate this threat, SecureJS requires the service provider to sign the JavaScript related data sent to the client platform.

6.2.1.10 Threat Nº 10

Definition. The service provider can repudiate the remote attestation related data (③) sent to the client platform.

Risk 1. The chance for this threat to happens is 7 (low) since during the accomplishment of this project it is assumed the good willing of the service provider, it can be assumed that the chances for the service provider to repudiate its own data is low. The damage generated by this attack is 7 (medium) since this threat could result into a problem, for example in the case that the remote attestation data affected the state of the client platform.
Mitigation. In order to mitigate this threat, the EPID protocol requires the service provider to sign the remote attestation related data sent and authenticate to the client platform.

6.2.1.11 Threat Nº 11

Definition. The client platform can repudiate the remote attestation related data (③) sent to the service provider.
Risk 1. The chance for this threat to happens is 7 (low) since it is assumed that the SGX technology guarantees security and fairness. The damage generated by this attack is 7 (medium) since the data affect in a insecure manner, i.e. the remote attestation related data affects the correctness of the service provider's device.
Mitigation. In order to mitigate this threat, the EPID protocol requires the SGX technology to sign the remote attestation data sent and authenticate to the service provider.

6.2.1.12 Threat Nº 12

Definition. The SGX technology can repudiate the result value (④) sent to the service provider.
Risk 1. The chance for this threat to happens is 7 (low) since it is assumed that the SGX technology guarantees security and fairness. The damage generated by this attack is 7 (medium) since this threat can become a problem, for example in the case that the result value affected the state of the service provider's device.
Mitigation. In order to mitigate this threat, SecureJS requires the SGX technology to sign the result value sent to the client platform.

6.2.2 STRIDE outcome

After applying STRIDE, the necessity of security techniques was obvious. Therefore, the design shown in Figure 6.12 was affected resulting in the graphical design shown earlier the thesis in Chapter 4 in Figure 4.1. Table 6.1 shows a summarized list with the threats described before, their risk level and whether they have been mitigated.

Table 6.1: Summary of threats reasoned via STRIDE threat modeling.

Threat number	Damage	Chance	Risk	Mitigation
1-7	10	1	10	Implemented
8	9	3	3	Implemented
9-12	7	7	1	Implemented

7
Discussion

This chapter includes the discussion resulting from the completion of the project. First, a number of facts and topics that were generated during the performance of the project are discussed. These discussions involve other approaches considered for the development of SecureJS (Chapter 7.1), the reasoning of the results generated by the tests (Chapter 7.2), and the comparisson between SecureJS and its direct opponent TrustJS (Chapter 7.3).

After that, three more sections are included; Chapter 7.4 contains the reasoning carried out around SecureJS in terms of sustainability, Chapter 7.5 describes the thoughts generated during the design of the project from an ethical point of view, and Chapter 7.6 gathers the ideas arose in order to continue the research done in this project or to improve the solution proposed, SecureJS.

7.1 Other approaches and limitations

During this project, two options have been considered as JavaScript interpreters; Chrome's V8 JavaScript engine [13] and MuJS JavaScript interpreter [9]. As explained in Chapter 5, MuJS has been the final selection for SecureJS. Nevertheless, V8 was considered since it is a sophisticated, optimized, and powerful interpreter capable of a great performance. V8 is a well known interpreter used by the Chrome web browser to execute JavaScript code. It also fulfills the newest standards for JavaScript execution. On the other hand, all those features require of a large amount of lines of code. This fact in combination with the restrictions of Intel SGX for I/O functionalities results in a situation of high complexity for embedding the V8 interpreter in an enclave.

A quick search for the SGX non-supported functions over the code files of both interpreters showed a large difference of positive matches (around 190,000 matches in V8 vs around 450 in MuJS). As mentioned in Chapter 5.4.3, the code of the interpreter has been modified in order to achieve an embeddable version than can work with the SGX technology. Therefore, the high number of matches that V8 contains and the limited time available for the implementation of this project, made the selection of the V8 engine as part of SecureJS impossible, due to the amount of work time required for adapting V8 to the SGX technology.

Another approach taken during the design of SecureJS included the usage of Graphene library OS [4]. Graphene is a lightweight guest operating system, similar to virtual machines, with virtualization benefits such as guest customization, platform independence and migration. Graphene can run applications in an isolated environment and it also supports the execution of applications in an SGX enclave. However, after some testing, it was found out that Graphene does not provide support for the remote attestation process designed by Intel, which is a key feature of SecureJS. Therefore, the usage of Graphene for SecureJS was dismissed.

7.2 Discussion of the results

In relation to the tests results, the graph in Figure 6.10 shows that the memory usage of the solution (note that SecureJS requires the usage of the Client application and the remote attestation Server) is higher than the one of the NodeJS server. However, the comparison between the parts delegated to the service provider, NodeJS and remote attestation Server, which is not an optimized version, show that the remote attestation Server has lower memory usage. This fact can support the decision of the service providers to make use of SecureJS as an alternative, in order to reduce the memory needs of the web application services.

In addition, both NodeJS server and the remote attestation Server have a base memory usage that must also be cosidered. Figure 6.11 show that NodeJS requires a higher base memory in order to work. However, many of the current web-application servers world-wide already have a NodeJS environemnte deployed and this memory usage is negligible in those cases.

Note that the tests were performed with individual clients making use of the web application in a serial setup (one request at a time). Hence, the memory usage of the NodeJS and the remote attestation Server can be affected by the increase of the number of clients requesting the services of the web application server at the same time. Nonetheless, the NodeJS server uses a higher quantity of memory compared to the remote attestation server (note that the last can be optimized). In contrast, the memory usage for the client-side devices stays stable per request for the client-side device no matter the number of clients connecting to the server. This way, the memory usage balance can be moved from the server and distributed among the clients, resulting in a substantial reduction for the server and a permissible increase in each client device.

In addition, all three macro-benchmark results represented in the Figures 6.1, 6.2, and 6.3 show that SecureJS requires of more time to execute JavaScript code compared to the commonly used setups (Chrome and NodeJS). This difference can be admissible in the low computational level code, however, it becomes a problem with high resource consumption code. Focusing on the results shown by Figure 6.6, it can be seen that most of the time consumed by SecureJS is used for the JavaScript component. Moreover, as shown in Figure 6.8, the fact that the JavaScript interpreter is embedded in an enclave does affect the performance since the execution time is higher to the simulations performed in a normal environment.

Note that, the time differences between the SecureJS mode and the normal environment simulations do not seem to be generated from the encryption and verification features that the JavaScript component has to carry out before and after the JavaScript execution. As Figure 6.7 shows, even when the data exchanged is large (a list of all prime numbers between 2 and 100K), the time required for signing, decryption (around 1 millisecond) and encrypting (around 7 milliseconds) are low.

7.3 Comparison between SecureJS and TrustJS

The comparison between SecureJS and TrustJS seems to be balanced towards the TrustJS implementation. If the results in Figure 6.9 are analyzed, it can be seen that there exits a time difference of around 400 milliseconds between both implementations. Looking to Figure 6.4, it can be determined that this difference can be due to the fact that SecureJS manages the creation and destruction of an enclave (Enclave Management is around 330 milliseconds) and accomplishes a remote attestation process (Remote Attestation component requires around 86 milliseconds) every time SecureJS is used.

On the other hand, as explained before in Chapter 3.4, David Goltzsche et al. designed an approach were a pool of enclaves is initialized at the start of the web-browser and an unique enclave is assigned per each tab opened by the user. In addition, a simplified version of the remote attestation process (the QUOTE is sent to the server via HTTP request) is done once right after assigning an enclave to a new tab. Therefore, the performance results for TrustJS [31] are not affected by the enclave management nor the remote attestation process.

In addition, David Goltzsche et al. also show the results gathered from performance tests carried out with TrustJS in terms of CPU usage. TrustJS seems to reduce the CPU load of the server significantly [31]. This resource has not been analyzed during this thesis, however, in the same manner as TrustJS does, it seems possible that SecureJS can also reduce the CPU load of the server-side device. Nevertheless, some performance tests are required in order to confirm this conjecture. This can lead to a future work that focuses on a deeper analysis of SecureJS.

7.4 Sustainability

SecureJS offers the developers a secure environment for code execution that can lead to better and more efficient services. The time used by the code developers to ensure the security aspects of their code can now be used to improve the efficiency and performance of the web services. Moreover, these efficient services can be accessed world-wide via the internet and, therefore, any person connected to the internet can take advantage of those benefits.

In addition, the possibility of workload distribution, that SecureJS promotes, allows the reduction of the dimension of the server parks owned by the server providers. This fact means a reduction in the costs for the service providers since less computational resources such as, CPU and memory, are needed in order to offer web-application services to the clients. This reduction in computational resources also means a reduction in the usage of the natural resources, such as metals and petroleum. Every piece of hardware component makes use of a large number of different finite natural resources. Moreover, the manufacturing of those natural resources into a useful product also produces contamination. Therefore, a reduction of the usage of these finite natural resources is key for a future sustainable world.

7.5 Ethics

The privacy of the parties taking part in a web-application structure, client and service provider, has been an important feature that has been supported during the design and implementation of SecureJS. In order to keep the privacy of the parties, SecureJS does not store nor analyze any of the data generated during the usage of SecureJS, such as web-pages accessed by the client, or data provided to SecureJS, such as JavaScript code sent by the service provider.

Last, as mentioned in Chapter 2.6, the service provided by SecureJS can be used by the service provider in order to make use of the execution power in the client-side device without any prior notification to the user. This situation has been assumed as impossible during the project by assuming an honest service provider. However, this problem exists and can be used by not that honest service providers. As a result of the reasoning done around this topic, a number of ideas that can be added to SecureJS were triggered.

The first approach is to notify the user every time a web-page is trying to make use of SecureJS. This way, the user can decide whether it is necessary or not to use the service provided by SecureJS. A similar approach is by adding a feature of white-listing, which defines the list of service providers that are allowed by the user to request the usage of SecureJS. A more complex approach is to implement a resource consumption management feature that can limit the execution resources allocated by the service provider to execute the JavaScript code.

7.6 Future work

As mentioned above, the main drawback of SecureJS is the time consumption in comparison with TrustJS and setups that make use of the V8 engine. Therefore, applying the enclave pool approach from TrustJS or modifying the V8 engine into a version supported by the SGX technology can be a good improvement for SecureJS. As a result, the enclave management time can be reduced or the code execution time can be improved.

In relation to Graphene, the developers have expressed their intentions of adding the feature of Intel's remote attestation process to their product. This fact would make Graphene become a viable option to be included in the design of an improved version of SecureJS. Moreover, the usage of Graphene can help with the embedding process of the V8 engine into an SGX enclave since many of the SGX non-supported I/O functions are handled and adapted to SGX by Graphene. Hence, the time work required for the adaption of V8 can be reduced.

Last, the approach taken in SecureJS can be used in other environments that hold a similar lack of trust problem. For example, Sancus [42] can make use of the SGX technology in the embedded devices. Additionally, other applications that provide data exchange services, such as email services and FTP services, can also use the approach taken to implement an integral and confidential service. The solution can also be improved by adding other language interpreters and can be used in applications that require secure remote execution of code.

8 Conclusion

In the current web-application infrastructure, there exists a lack of trust for the execution of JavaScript code in client-side devices (Chapter 2). This problem is a result of the absence of process monitoring and attestation technologies in web-applications. Therefore, the server-side device is forced to validate and verify every value generated in the client-side device.

To mitigate this problem, we introduce SecureJS, a proof of concept implementation that aims to provide integrity and confidentiality for JavaScript code execution and result output in the client-side device for Chrome web-browser users (Chapter 4).

SecureJS promotes a trust relation for web-application services that require the execution of JavaScript code. SecureJS is able to provide secure guarantees for the code execution (Chapter 5). Moreover, the proof of concept also provides security guarantees for the data exchanged between the server-side device and SecureJS. We demonstrate the correctness of SecureJS by accomplishing a case study through simulations of real world web-applications and security evaluations carried out for SecureJS (Chapter 6).

To sum up, SecureJS offers a light and secure environment for web-application services. A new branch of implementation possibilities are provided for web developers that were limited by the no-trust relation between the client-side and server-side devices. Furthermore, the possibilities are not only related to the area of security. SecureJS also provides the opportunity of expanding the area of resource consumption efficiency by providing a flexible way to balance the work load.

Bibliography

[1] Chrome extensions - Native Messaging. https://developer.chrome.com/extensions/nativeMessaging [Accessed: February 2017].

[2] Chrome extensions - Overview. https://developer.chrome.com/extensions/overview [Accessed: February 2017].

[3] Desktop Browser Version Market Share. https://www.netmarketshare.com/browser-market-share.aspx?qprid=2&qpcustomd=0 [Accessed: February 2017].

[4] Graphene library OS. https://github.comsecureJSGit/oscarlab/graphene/wiki [Accessed: February 2017].

[5] Intel Software Guard Extensions Remote Attestation End-to-End Example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example [Accessed: April 2017].

[6] Introducing JSON. http://www.json.org/ [Accessed: April 2017].

[7] Linux's manual page for command: TIME. http://man7.org/linux/man-pages/man1/time.1.html [Accessed: June 2017].

[8] Message Passing. https://developer.chrome.com/extensions/messaging [Accessed: February 2017].

[9] MuJS, lightweight implementation of the Javascript language in a library . https://mujs.com/ [Accessed: March 2017].

[10] Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. https://nodejs.org/en/about/ [Accessed: June 2017].

[11] Pico Process Overview. https://blogs.msdn.microsoft.com/wsl/2016/05/23/pico-process-overview/ [Accessed: March 2017].

[12] Threat Risk Modeling: STRIDE. https://www.owasp.org/index.php/Threat_Risk_Modeling#STRIDE [Accessed: February 2017].

[13] V8, Google's high performance, open source, JavaScript engine., date = 2017. https://developers.google.com/v8/ [Accessed: March 2017].

[14] Hype Cycle for Infrastructure Strategies, 2016. Technical report, Gartner, Stamford, USA, 2016.

[15] ICT Facts and Figures 2016. Technical Report 1, International Telecommunication Union, Geneva, Switzerland, 2016.

[16] Intel® Software Guard Extensions: EPID Provisioning and Attestation Services. Technical report, Intel Corporation, 2016.

[17] Protect Application Code, Data, and Secrets from Attack. Technical report, Intel Corporation, 2016.

[18] State of software security 2016. Technical report, Veracode, Burlington, Massachusetts, USA, 2016.

[19] Web applications security statistics report 2016. Technical report, WhiteHat Security, Santa Clara, California, USA, 2016.

[20] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing.

[21] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves. Technical Report 2017/5, Imperial College London, March 2017.

[22] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, Aug. 2015.

[23] S. A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, Cambridge, MA, USA, 2000.

[24] E. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 132–145, New York, NY, USA, 2004. ACM.

[25] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. *SIGPLAN Not.*, 48(4):253–264, Mar. 2013.

[26] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: A Hardware-software Approach to Full System Security. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 108–119, New York, NY, USA, 2011. ACM.

[27] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. *SIGPLAN Not.*, 49(4):81–96, Feb. 2014.

[28] P. Dewan, D. Durham, H. Khosravi, M. Long, and G. Nagabhushan. A Hypervisor-based System for Protecting Software Runtime Memory and Persistent Storage. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 828–835, San Diego, CA, USA, 2008. Society for Computer Simulation International.

[29] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1 edition, June 2011.

[30] A. R. Fernandez. SecureJS git repository. `https://github.com/AsierRF/SecureJS.git`.

[31] 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*, EuroSec'17, pages 7:1–7:6, New York, NY, USA, 2017. ACM.

[32] D. Gries and J. Misra. A Linear Sieve Algorithm for Finding Prime Numbers. *Commun. ACM*, 21(12):999–1003, Dec. 1978.

[33] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Commun. ACM*, 52(5):91–98, May 2009.

[34] W. G. J. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 174–183, New York, NY, USA, 2005. ACM.

[35] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end Security via Automated Full-system Verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 165–181, Berkeley, CA, USA, 2014. USENIX Association.

[36] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. *SIGPLAN Not.*, 48(4):265–278, Mar. 2013.

[37] H. Krawczyk. *"SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols"*, pages 400–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[38] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 511–526, Washington, D.C., 2013. USENIX.

[39] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. MiniBox: A Two-way Sandbox for x86 Native Code. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 409–420, Berkeley, CA, USA, 2014. USENIX Association.

[40] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[41] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 28–37, New York, NY, USA, 2012. ACM.

[42] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 479–494, Berkeley, CA, USA, 2013. USENIX Association.

[43] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser Analysis of Web-based Malware. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets*, HotBots'07, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[44] P. Stewin and I. Bystrov. *"Understanding DMA Malware"*, pages 21–41. Springer, Berlin, Heidelberg, 2013.

[45] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *23rd USENIX*

Security Symposium (USENIX Security 14), pages 655–670, San Diego, CA, 2014. USENIX Association.

[46] S. Weiser and M. Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. *CoRR*, abs/1701.01061, 2017.

[47] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Communications of the ACM*, 53(1):91–99, 2010.

A

Appendix

This appendix contains all the additional information related to the thesis. The aim of this chapter is to provide extra information that can be interesting for the reader but is not necessary to comprehend the document. The following sections provide more information related to topics mentioned in the document.

A.1 Remote attestation extended

The functionality of remote attestation is provided by the Intel SGX technology to ensure the trust over a platform (*platform* will be used within this section to refer to the device making use of the SGX software) running this technology. This is an important feature for the actual situation of computers, where the exchange of data and service provision over a network is important. Before explaining the full process of remote attestation it is necessary to understand some relevant definitions and setup requisites.

Root Provisioning Keys. Keys generated and stored by Intel used for provisioning purposes. These keys are unique for each platform and are hardcoded in the microprocessor during the fabrication process.

Root Sealing Keys. Keys generated but not stored by Intel used for sealing purposes. These keys are unique for each platform and are hardcoded in the microprocessor during the fabrication process.

Trusted Computing Base. The TCB is composed by the CPU and the content inside its packages (hardware logic, microcode, registers, cache memory) and some special SGX software components, such as the ones used for attestation, *quoting enclave* and *provisioning enclave*.

Enhanced Privacy Identifier (EPID). EPID is the algorithm designed by Intel for SGX attestations.

EPID Group Master Key. Key used by the *Provisioning Service* and the platform in order to generate the *Attestation key*.

EPID Group Public Key. Key used by the *Attestation Service* in order to verify the *quote*'s data.

Provisioning Service. Service offered by Intel in order to exchange the information related to the remote attestation (Attestation key, *TCB* correctness demonstration, etc.).

Attestation Service. Service offered by Intel in order to verify the *Quote*.

Attestation key. A cryptographic asymmetric private key used by the *Quoting enclave* to sign the *Quote*.

Report. The report is a structure that contains the two identities of the enclave, attributes associated with the enclave, the trustworthiness of the hardware TCB, a Message Authentication Code (MAC) tag and some additional and optional information.

Quote. The quote is a similar structure as the Report that, instead of a MAC value, contains a signature of the structure. This signature is done within the *Quoting enclave* using the *Attestation key.*

Quoting enclave. A special SGX enclave completely focused to remote attestation. This enclave is the only one that has access to the Attestation key and includes the required functionalities to generate the Quote from the Report.

Provisioning enclave. A special SGX enclave that focuses on the provisioning process.

The EPID algorithm is based on the Direct Anonymous Attestation (DAA) algorithm [24]. It shares the privacy enhancing properties of DAA and adds a revocation mode in order to avoid the usage of compromised keys and signatures. This algorithm requires Intel to provide a full architecture of servers as explained in [20]. Figure A.1 shows a representation of that architecture.

In this architecture, the *Offline Key Generation* facility is responsible for creating and storing the *Root Provisioning Keys* and the generation of the *TCB* verification data and the EPID Master Keys that are passed to the *Provisioning Service* for attestation purposes and the *EPID Group Public Key* that are passed to the *Attestation Service* for verification purposes. The *High Volume Manufacturing* facility is responsible for generating the *Root Sealing Keys* and hardcoding both, the *Root Provisioning Key* and the *Root Sealing Key* into the Intel microprocessors.

A.1.1 Setup for remote attestation

The architecture created by Intel provides two services, *Provisioning Service* and *Attestation Service*. In addition, it makes use of pairs of asymmetric keys (*EPID Group Master Key* and *EPID Group Public Key*) divided into groups. Before any remote attestation process can be performed, it is required that the SGX software is setup for remote attestation. Therefore, the first time the SGX software is deployed in the platform, the platform communicates with the *Provisioning Service* in order to demonstrate that it is executing inside an enclave on an Intel platform with a particular *TCB*. During the same process, the *Attestation key* is generated by the *Provisioning enclave* in cooperation with the *Provisioning Service* as a derivation of an *EPID Group Master Key.*

As a result, the data signed by the *Attestation key* can be verified by the *Attestation Service* via the *EPID Group Public Key* of that particular group. The reason for Intel to separate the master keys in groups is to allow the verification of the data while ensuring the privacy of the platform. This means that the *Attestation Service* is able to verify the *Quote* signed by the *Attestation key* using the particular group's *EPID Group Public Key* but is not able to link that data to a specific member of the group.

On the other side of the remote attestation schema, the third party (named as *Service Provider* in advance) that is willing to verify the correctness of the platform

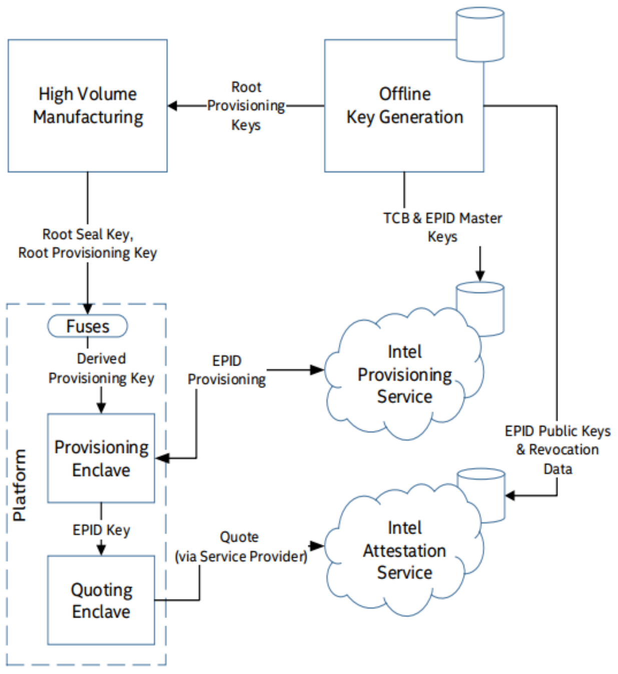

High Volume Manufacturing
Root Provisioning Keys
Offline Key Generation
Root Seal Key, Root Provisioning Key
TCB & EPID Master Keys
Fuses
Derived Provisioning Key
EPID Provisioning
Intel Provisioning Service
Provisioning Enclave
Platform
EPID Public Keys & Revocation Data
EPID Key
Quote (via Service Provider)
Intel Attestation Service
Quoting Enclave

Figure A.1: SGX infrastructure for Enhanced Privacy Identifier (EPID) algorithm. Image adapted from [20].

(the web application server in this project) has to obtain a signed certificate from a recognized certificate authority and register it in order to get a service provider ID (SPID) from Intel. This way, the service provider will be allowed to make use of the *Attestation Service* provided by Intel. In addition to that, the third party needs to deploy an attestation service that is able to follow the attestation process as explained in the following Chapter A.1.2.

The remote attestation protocol is based on the Sigma protocol with small differences. The Sigma protocol [37] that is used in Internet Key Exchange (IKE) v1 and v2 makes use of Public Key Infrastructure (PKI) in both the client and server side devices for authentication. On the other hand, Intel's protocol keeps the PKI in the service provider, while uses EPID for the authentication of the platform. In addition, the service provider changes the RSA algorithm for the Elliptic Curve Digital Signature Algorithm (ECDSA) to generate the key pair used in the authentication part of the protocol and Elliptic Curve Diffie–Hellman (ECDH) for the actual key exchange.

The architecture developed by Intel is a complex sum of parts that aims to provide the required security resources (keys, signatures, etc.) while providing privacy and fairness. A complete explanation of each part would require a large explanation. However, this report contains a simplified description of the EPID architecture that provides the required information to properly understand the following content. Therefore, the reader is encouraged to consult more specialized resources, such as [20] and [16], in order to get more information about SGX.

A.1.2 Remote attestation process

Once the SGX software is properly setup and the third party has fulfilled the registration and service deployment requisites, the resultant communication schema looks like shown in Figure A.2. With all the parts that take place in the process, the remote attestation process can be accomplished as follows.

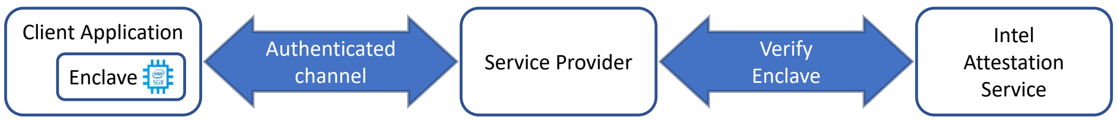


Figure A.2: SGX remote attestation communication schema (high level view). Image adapted from [5].

Remote Attestation makes use of a modified Sigma protocol to facilitate a Diffie-Hellman Key Exchange (DHKE) protocol between the client and the server. The shared key obtained from this exchange can be used to encrypt secrets to be provisioned to the enclave. The client enclave would then be able to retrieve the same key and use it to decrypt the secret. In order to achieve that and the platform correctness verification, the communication schema shown in Figure A.3 should be followed.

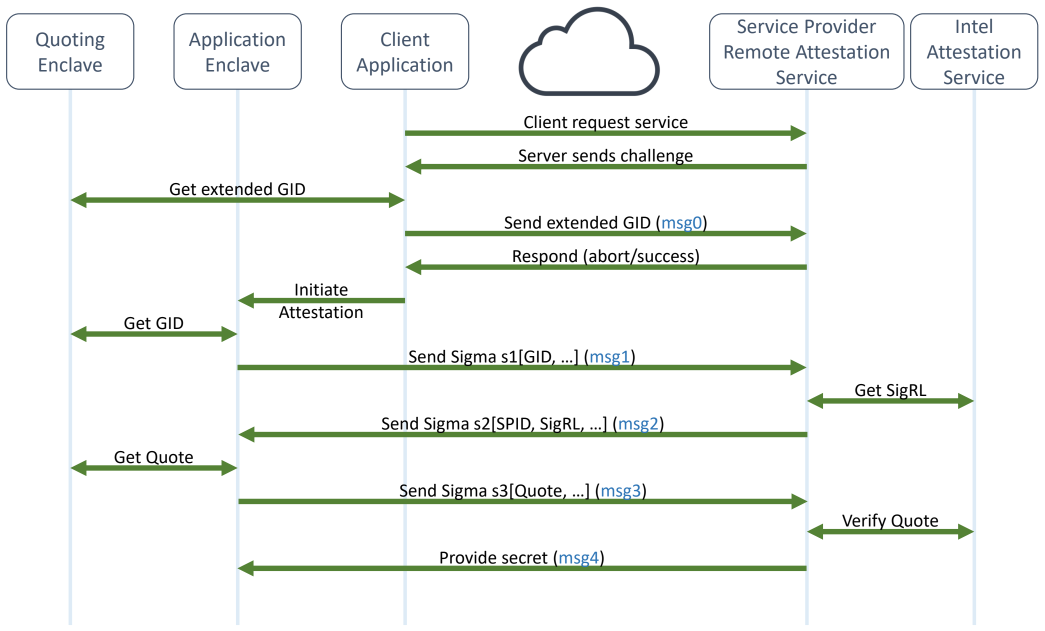


Figure A.3: SGX remote attestation communication schema (low level view). Image adapted from [5].

First, the client starts the communication process with the service provider by sending a service request message. To which, the service provider answers with a challenge. At this point, the communications for the remote attestation process is started and the following messages can be exchanged.

MSG0. In response to the challenge request, the client performs some steps to build the content of the initial message in the remote attestation process. First, the client initializes the *Quoting Enclave*, which accepts the service provider's challenge and initializes a DHKE protocol for the coming messages. Second, it requests the Extended Group ID (GID) of the *EPID* to the *Quoting Enclave*. Third, the client generates the body of the msg0 including the Extended GID. Last, it sends the msg0 to the service provider. ***Note:*** The format of the response to msg0 is delegated to the service provider. Nevertheless, the service provider should verify the Extended GID received and send an abort or success response based on the output of that verification. The *Attestation service* only supports the value of zero for the EGID.

MSG1. Upon a success response from the service provider, the client initiates the attestation process within the platform enclave. First, the enclave requests the GID value to the *Quoting enclave* and makes use of the SGX functionalities to safely calculate the client's public key for the DHKE. Second, the enclave puts the GID and the public key for the DHKE in the body of msg1. Last, the client sends the msg1 to the service provider.

MSG2. Given the reception of msg1 from the client, the service provider performs a number of tasks. First, it checks the values in the body of msg1. Second, the service provider generates its own DHKE parameter. Third, it sends a query to the *Attestation Service* to gather the Signature Revocation List (SigRL is a list related to the signature based revocation mode added by Intel to the DAA algorithm). Fourth, the service provider adds the DHKE parameter and the SigRL to the body of msg2. Last, sends msg2 to the client.

MSG3. Once the client receives the msg2, the following tasks are performed. First, it checks the SigRL and the signature of the service provider. Second, the client gathers the *Quote* from the *Quoting enclave*. Third, ti adds the *Quote* to the body of msg3. Last, the client sends msg3 to the service provider.

MSG4. Upon receiving msg3 from the client, the service provider first checks the DHKE parameters and the enclave specific parameters embedded in the *Quote*. Second, it forwards the Quote and signature to the *Attestation Service* for verification. Third, upon a successful verification response from the *Attestation Service*, the service provider builds the body of msg4 by adding the attestation status and some optional values, such as a secret value. At this point, the DHKE protocol is finished and both sides share a symmetric key, which allows the msg4 to be encrypted.

A.2 JSON for SecureJS

The following Table A.1 contains the list of allowed JSON keywords. In addition, the table describes the functionality of each keyword and denotes the SecureJS modes in which the keyword can be used. Listings A.1, A.2, A.3, A.4, and A.5 show examples of the messages exchanged between the web-page and SecureJS.

```
{
  "ENCRYPTION": "Encryption of {JavaScript code;Main function} in
     hexadecimal format",
  "SIGNATURE": "Signature of {JavaScript code;Main function} in
     hexadecimal format",
  "VARIABLE": [
    {"TYPE":"int","ORDER":"0","VALUE":"User input variable 1"},
    {"TYPE":"int","ORDER":"1","VALUE":"User input variable 2"}
  ]
}
```

Listing A.1: JSON format message example for encryption mode with two variables

```
{
  "ENCRYPTION": "Encryption of {JavaScript code;Main function} in
      hexadecimal format",
  "SIGNATURE": "Signature of {JavaScript code;Main function} in
      hexadecimal format"
}
```

Listing A.2: JSON format message example for encryption mode with no variables

```
{
  "VALUE": "Result value generated from the JavaScript execution in
       plaintext format",
  "SIGNATURE": "Signature of VALUE in hexadecimal format"
}
```

Listing A.3: JSON format response message example for signature mode

```
{
  "ENCRYPTION": "Encryption of the result value generated from the
      JavaScript execution in hexadecimal format",
  "SIGNATURE": "Signature of the result value (VALUE in singatue
      mode) in hexadecimal format"
}
```

Listing A.4: JSON format response message example for encryption mode

```
{
  "ERROR": "Error text"
}
```

Listing A.5: JSON format response error message example

Table A.1: Extended list of allowed JSON keywords, description and usage.

JSON keyword	Reference	Mode
VARIABLE	The array that contains the variables	Both modes (optional value)
CODE	JavaScript code in plaintext	Signature mode
SIGNATURE	Signature of the CODE in hexadecimal format or the signature of the result value from the JavaScript code	Both modes
MAINFUNCTION	Name of the main function of the CODE for the signature mode	Signature mode
ENCRYPTION	Encryption of the CODE in hexadecimal format or the encryption of the result value from the JavaScript code	Encryption mode
TYPE	Type of a variable within the VARIABLE array	Both modes (mandatory if VARIABLE included)
ORDER	Order of a variable within the VARIABLE array	Both modes (mandatory if VARIABLE included)
VALUE	Value of a variable within the VARIABLE array or the result value from the JavaScript code	Both modes (mandatory if VARIABLE included)
ERROR	Error message sent by the solution	Both modes