TrustJS: Trusted Client-side Execution of JavaScript

David Goltzsche
TU Braunschweig, Germany
goltzsche@ibr.cs.tu-bs.de

Colin Wulf
TU Braunschweig, Germany
cwulf@ibr.cs.tu-bs.de

Divya Muthukumaran
Imperial College London, UK
d.muthukumaran@imperial.ac.uk

Konrad Rieck
TU Braunschweig, Germany
k.rieck@tu-bs.de

Peter Pietzuch
Imperial College London, UK
prp@imperial.ac.uk

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

ABSTRACT

Client-side JavaScript has become ubiquitous in web applications to improve user experience and reduce server load. However, since clients are untrusted, servers cannot rely on the confidentiality or integrity of client-side JavaScript code and the data that it operates on. For example, client-side input validation must be repeated at server side, and confidential business logic cannot be offloaded. In this paper, we present TRUSTJS, a framework that enables trustworthy execution of security-sensitive JavaScript inside commodity browsers. TRUSTJS leverages trusted hardware support provided by Intel SGX to protect the client-side execution of JavaScript, enabling a flexible partitioning of web application code. We present the design of TRUSTJS and provide initial evaluation results, showing that trustworthy JavaScript offloading can further improve user experience and conserve more server resources.

Keywords

Trusted Computing, Trusted Clients, Intel SGX, JavaScript

1. INTRODUCTION

Web applications increasingly replace traditional desktop applications, mainly because developers gain instant deployability and platform independence. With single-page applications [27], developers utilise client-side JavaScript code to provide a user experience similar to desktop applications. This resulted in a substantial growth of JavaScript, manifesting for example in JavaScript being the most popular language on GitHub for several years [2] and Google enabling their crawlers to also handle JavaScript [18].

The success of JavaScript is founded in the ability to offload computations to clients in a standardised, widely accepted fashion. Offloading has two advantages: Firstly, it minimises round trip times, resulting in a better user experience. Secondly, service providers can save costs as less resources are demanded from servers.

However, despite all these benefits, offloading code to clients has a strong limitation: From the provider's view, clients can *never* be assumed to be trustworthy. As a consequence, computation results from clients may be wrong, corrupted, or incomplete. To solve this, client results are typically checked for consistency at the server, which often requires recomputation [35]. An example for this is input validation; server software repeats computations already performed on client-side. Considering this on a global scale, this is a massive waste of resources. Additionally, related projects [9, 1] show that validation of input values on client and server side can be error prone and introduce vulnerabilities.

Furthermore, the intellectual property (IP) of confidential JavaScript code offloaded to the client cannot be protected. Companies may have decided against client-side computation because this would require the disclosure of algorithms or business logic. For example, banks might want to keep their scoring algorithms for credit checks confidential. A contemporary approach to resolve this issue is *code obfuscation* [14, 23], which is said to prevent theft, reverse engineering or copyright infringements. However obfuscation has been proven as insufficient [32, 13, 24].

These limitations, however, can be circumvented by exploiting novel hardware extensions that enable trusted execution, such as *Intel SGX*. The Intel Software Guard Extensions (SGX) consist of a number of additional x86 instructions that allow the creation of so-called *enclaves*. In these enclaves, hardware mechanisms protect the execution of code from other potentially privileged code, such as the operating system, the kernel and drivers. Moreover, SGX prevents certain classes of hardware-based attacks by utilising memory encryption and integrity checks.

SGX has recently been utilised for different client-side solutions such as password stores [10, 25] and anti-cheat software [7]. Additionally, it can enable secure execution of nearly or even totally unmodified legacy applications [5, 34, 12, 8], help securing pub/-sub systems [31] and coordination services [11]. However, none of these works addresses client-side web applications enabling trusted execution of JavaScript code within commodity browsers.

In this paper we present the TRUSTJS web development framework. It allows the development of web applications that execute trusted and untrusted standard client-side JavaScript code side by side. This is achieved, by adding a *trusted JavaScript engine* to a browser and protect it with SGX. Being transparently integrated into Firefox as an add-on, both installation and usage is easy for end users. For developers, TRUSTJS provides techniques to simplify the process of writing *trusted JavaScript*. Additionally, TRUSTJS isolates the execution of JavaScript in browser tabs at runtime, distinguishing between untrusted and trusted parts. We present an initial evaluation, which shows the benefits of TRUSTJS in terms of latency and scalability, thus supporting our vision of a new way of developing web applications.

2. BACKGROUND AND THREAT MODEL

In this section, we give details about Intel SGX, the core technology behind TRUSTJS and introduce our assumed threat model.

2.1 Intel SGX basics

In a nutshell, Intel SGX enables the protection of data and computation in compartments called *enclaves*. Plaintext of enclave-protected data is only available for computation inside the CPU package and is guarded against unauthorised accesses by privileged code or physical attackers. To achieve this, Intel SGX extends the instruction set for recent x86 processors to enable the creation and management of enclaves. Enclaves are associated with an isolated logical memory range inside the address space of applications. SGX protects the confidentiality and integrity of this range with check sums and memory encryption. The logical addresses of the enclave are mapped to pages, stored in a system-reserved memory range called Enclave Page Cache (EPC).

Enclave development and interaction. Applications interact with enclaves via *ecalls* and *ocalls*. These calls are defined by the developer on function granularity in the *enclave description language*. Figure 1 shows the basic interaction pattern with enclaves: First, the application creates the enclave from a binary. After that, ecalls can be made into the enclave, and ocalls from the enclave; both returning to the caller function after completion. Intel provides a Software Development Kit (SDK) [22] for defining and handling ecalls, ocalls and the enclave's life cycle.

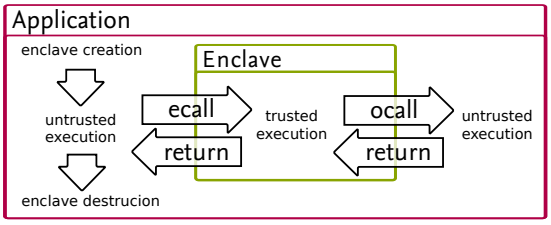


Figure 1: Basic interaction pattern between application and enclave with ecalls and ocalls.

Local and remote enclave attestation. A key concept of SGX is attestation, which enables authentication of enclaves to a local or remote party. It is based on SHA-256 digests called *identities* or *measurements* and supported in two variants: `MRENCLAVE`, identifying a specific version of an enclave, and `MRSIGNER`, identifying the creator of the enclave. Moreover, an attestation is *local* if the enclave is authenticated to another enclave on the same platform, whereas *remote attestation* extends this local mechanism to allow a remote party called the *challenger* the authentication of enclaves [4]. TRUSTJS heavily relies on this feature to create a trusted channel between providers and enclaves on client machines, as described in §4.2.

Both measurement variants are based on data structures called *reports* containing both forms of enclave identities. While for local attestation, reports are generated for and verified by arbitrary enclaves, remote attestation relies on the Quoting Enclave (QE) issued by Intel. The QE locally verifies reports and transforms them into *quotes*, by adding a signature based on a key provisioned by Intel. When the challenger receives the quote, she sends it to the Intel-operated service called Intel Attestation Service (IAS), which checks the signature and returns the result. On a positive result, the challenger has successfully authenticated the remote enclave.

To perform a remote attestation, the challenger has to register for the IAS once. He will be assigned a unique Service Provider ID (SPID), which has to be transferred with every verification request.

Limitations of SGX. Despite all its benefits, the use of SGX imposes some limitations, that should be considered during system design: first, applications running in enclaves have to be *operating system agnostic*: by design, enclaves do not depend on mechanisms usually provided by the operating system such as system calls and memory management. Nevertheless, providing system calls in enclaves is possible [5, 12, 34].

Second, the EPC is limited to 128 MB, a small fraction of memory that today's applications consume. While page swapping is supported, recent publications have shown a higher memory demand results in a severe performance penalty [5, 11]. Thus, the memory demand of enclaves should be kept low. This also means, a system deploying enclaves at clients instead of servers has more resources available since the EPC is limited per machine.

2.2 SGX-aware threat model

We define three roles that interact when TRUSTJS is in operation: The *enclave builder*, the *user* and the *service provider*. The enclave builder creates the enclave and ships it with the browser add-on, which is installed by the user who owns and controls a client machine. Both the user and service provider trust the enclave and we assume they can check its publicly available implementation. The service provider offers a web-based service and uses TRUSTJS for trusted execution at the user's machine. He wants to protect the integrity of the JavaScript code and, in case of IP protection, also the confidentiality. The user trusts the service provider, in terms of the offered service being benign. However, he does not trust the (possibly encrypted) JavaScript code, but relies on the enclave for isolated and secure execution. This implies, JavaScript being unable to break out of the enclave.

For the design of TRUSTJS we act on a threat model typical for SGX enclaves: an attacker has the client machine under full control, including the operating system and the hardware—with the exception of the CPU. However, we exclude denial-of-service (DoS) attacks on enclaves from our threat model since the design of SGX allows the host operating system (OS) to control enclave's life cycles anyway. That means, attackers can prevent or abort the execution of enclaves, but should not gain any knowledge by doing so.

Furthermore, side-channel attacks [37, 36] based on vulnerabilities of the application running inside the enclave are not of interest: We assume all enclave software to be free of security-relevant bugs. Finally, we assume the design and implementation of SGX itself, including all cryptographic operations is secure and does not contain any vulnerabilities.

3. DESIGN OF TRUSTJS

Running trusted and untrusted JavaScript side by side in a browser requires changes to its architecture. Keeping the threat model described in § 2.2 in mind, we place an additional JavaScript interpreter inside an enclave and integrate it as a browser add-on. We rely on Intel SGX due to its availability on current commodity x86 processors, which will most likely be supported by a major fraction of future client machines. We refer to the SGX-protected JavaScript engine as *interpreter enclave*.

The high-level architecture of TRUSTJS is shown in Figure 2: A browser with the TRUSTJS add-on installed, having multiple instances of the interpreter enclave (IE), each one associated with a service provider (SP). TRUSTJS creates a *trusted channel* between the service provider and its assigned enclave, enabling them to communicate integrity and confidentiality protected.

The interpreter enclave should be viewed as a trusted *two-way sandbox* like Ryoan [21]: as in normal browsers, users trust the interpreter, in terms of JavaScript code not being able to break out of it. In addition, service providers assume that enclaves do not accept other JavaScript which could leak secrets. Service providers

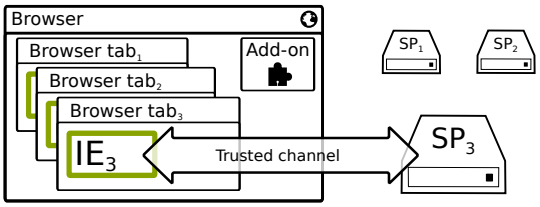


Figure 2: High Level Architecture of TrustJS; example with three service providers (SP), each associated with an client-side interpreter enclave (IE) running inside the browser.

can load arbitrary JavaScript code into interpreter enclaves, as long as the boundary between trusted and untrusted parts is on function granularity. We refer to JavaScript functions loaded into the enclave as *trusted functions*.

The add-on manages life cycles for multiple interpreter enclave instances: when the user opens a new tab, an enclave is assigned to it; when it is closed the enclave is destroyed. TRUSTJS uses an *enclave pool* containing more enclaves than open tabs. Enclaves are continuously created in the background, as the user opens new tabs. Thus, enclave initialisation times do not affect user experience. TRUSTJS ensures that internal states of the interpreter enclave can never be leaked to other providers, neither by reusing enclave instances nor by loading potentially malicious code by other parties. This is achieved by assigning new enclaves, if a provider change is detected within a tab. Also, the interpreter enclave is designed to load code signed by one single provider only.

For loading JavaScript functions into the enclave, service providers can choose between *authenticated* and *confidential* code, by either signing or additionally encrypting parts of the HTML files they deliver (see §5). As dynamically generated JavaScript cannot by authenticated, TRUSTJS does not support the commonly misused `eval()` function. Also, since encrypted JavaScript cannot be inspected, TRUSTJS offers policies, allowing users to decide for every website whether confidential JavaScript is enabled. This will influence the functionality of websites, but empowers users to prevent untrusted providers from executing confidential code on their machines. In addition, TRUSTJS guarantees the integrity and/or confidentiality of return values. Instead of returning plain values, JSON objects are returned. These contain the original return value, either in plaintext or encrypted as well as a Keyed-Hash Message Authentication Code (HMAC) over the data.

4. IMPLEMENTATION OF TRUSTJS

This section gives details about the implementation of TRUSTJS on a lower level. This comprises the client-side components (§4.1) and the remote attestation of interpreter enclaves (§4.2).

4.1 TrustJS client-side components

TRUSTJS consists of three components, as depicted in Figure 3: the TRUSTJS add-on installed in a browser, an *untrusted enclave bridge* and the interpreter enclave. All components are running in the browser process context, thus circumventing expensive IPC calls and being the easiest approach with Intel SGX. The components are interconnected with different mechanisms, either provided by the Mozilla Add-on SDK [28] or the Intel SGX SDK [22]: the browser and the add-on use *port objects* to communicate via asynchronous messages. Since the add-on and bridge are written in different languages we use *js-ctypes* [29] to create an appropriate interface between JavaScript and native C code. The bridge uses ecalls to enter the enclave and handles ocalls originating from it.

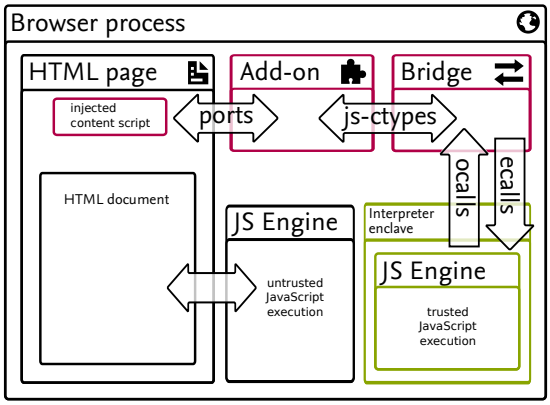


Figure 3: The components of TrustJS at client-side, showing untrusted components in red and trusted components in green.

In the following paragraphs, we describe all components in more detail and provide an overview of their tasks.

TRUSTJS browser add-on. The add-on is responsible for connecting the rest of TRUSTJS's components with the browser and initiating the transfer of trusted functions into the enclave. This is achieved by injecting a *content script* into pages loaded by the browser. The script finds all HTML `script` tags with `trustjs` attributes (see § 5) in the current document. As these contain signed or encrypted JavaScript code, they are transferred into the enclave for signature checks and/or decryption and later execution. Furthermore, the original trusted scripts are replaced with *proxies* that delegate calls to the implementations now residing in the enclave. This includes the handling of function parameters and return values.

Untrusted enclave bridge. The untrusted enclave bridge is a binary shipped with the add-on, implementing functions called by the extension. It is completely stateless, simply translating calls from the extension into ecalls as well as forwarding ocalls. In essence, the bridge exposes the interface of the enclave to the browser. Note, that the bridge cannot be omitted by letting the add-on call enclave functions directly, as the bridge needs to invoke SGX instructions necessary to start, enter and destruct the interpreter enclaves.

Interpreter enclave. The implementation of the interpreter enclave is also shipped with the add-on. Its purpose is the trusted execution of JavaScript. Our prototype is based on *MuJS* [6], a lightweight JavaScript interpreter that is fully compatible to version ES5 of the ECMA-262 standard [17]. Compared to more sophisticated interpreters like V8 [19] or SpiderMonkey [30], MuJS has a relatively small codebase of less than 14 KLOC and does not support just-in-time compilation. Both properties ease the process of porting it into the enclave and lead to a small trusted computing base, which improves security.

4.2 Remote attestation and key exchange

Service providers need a public key from and a shared secret with an assigned interpreter enclave for two crucial tasks: The signing and/or encryption of trusted functions and the verification/decryption of return values. In order to trust results, service providers need to ensure the aforementioned public key has been generated in an interpreter enclave they trust. Assuming the code of the interpreter enclave has been released as open source, service providers can check the implementation and maintain a list of hashes of trustworthy enclaves. By using compilation instructions by the enclave

builder for reproducible and verifiable builds [15], these hashes can be compared with the enclave measurement MRENCLAVE.

To establish trust between a service provider and an interpreter enclave, a remote attestation is performed. Depending on the use case, this happens before verifying return values or deploying confidential code. As described in § 2.1, the attestation is based on reports and quotes. By design, both report and quote can contain 512 bits of arbitrary data, which is included in all cryptographic operations performed on these data structures. We use this field to embed a hash of an ephemeral public key created during enclave start. By doing so, service providers can receive a public key from interpreter enclaves and a proof, that it has been generated by an enclave they trust.

The process for remote attestation and key exchange works as follows: the service provider embeds his SPID (obtained from Intel, see §2.1) and a target URL for the final quote into his document. TRUSTJS persistently caches SPIDs and URLs associated with providers to speed up setup time for future requests. When the user opens the browser (or a new tab), the add-on assigns an existing interpreter enclave, which has already generated an ephemeral key pair. As the user visits the providers website, the SPID is obtained from either the document or the cache. Subsequently, the add-on issues an ecall, creating a report, containing a hash of the generated public key. Upon return, the ecall discloses two data structures: the aforementioned report and the public key itself. Via another ecall to the QE, a quote based on the report and SPID is created. Note, that the quote also contains the hash of the public key and is verifiable by Intel. Subsequently, the TRUSTJS add-on sends both the quote and the public key to the URL defined by the service provider.

Every service provider operates a web service, which receives both the quote and public key. After the quote has been sent to the IAS and a response has arrived, the provider verifies three properties: (i) the IAS response is positive (ii) the public key matches the hash embedded in the quote (iii) the MRENCLAVE value in the quote matches a known hash. If these are fulfilled, the provider is certain a trustworthy environment has been created, the public key was generated in it and the corresponding private key has never left it. He generates a secret key, encrypts it with the public key and sends it back to the client.

After completion, the client uses the shared secret to add HMACs to or perform encryption on results of the trusted JavaScript functions. In contrast, the service provider uses the verified public key to encrypt or sign JavaScript code. In order to avoid additional message overhead, this message exchange is embedded in HTTP requests and responses that need to happen anyway for web applications to work (see Figure 4).

5. APPLICATION DEVELOPMENT

With the TRUSTJS framework, new possibilities are introduced to web developers as clients can execute JavaScript in a trusted way. Hence, a new programming paradigm is necessary. Developers should view client-side JavaScript as a central and trusted part of their infrastructure providing reliable computing resources. To allow this, developers need to perform additional steps: meta information has to be provided, trusted scripts have to be annotated and signed or encrypted. This section gives an overview about these steps and shows how TRUSTJS implements them.

For the remote attestation of the interpreter enclave (see §4.2) to work, developers have to provide additional information: first, clients need to know the SPID and a quote target URL for remote attestation (see § 2.1). These have to be embedded by developers as shown in Listing 1. When a page is loaded, the TRUSTJS add-on extracts this information.

```
<meta name="trustjs:spid" content="BF..40"/>
<meta name="trustjs:url"
    content="https://host/quote"/>
```

Listing 1: Meta information embedded by service providers for remote attestation.

```
<script trustjs-encrypt="yes">
  /* @exposed confidentialFunction 1 */
  function hiddenFunction(y) { ...
  }
  function confidentialFunction(x) { ...
    hiddenFunction(x);
  }
</script>
<script>var a = confidentialFunction(42);</script>
```

Listing 2: Trusted JavaScript code before encryption.

Second, developers have to annotate JavaScript code for trusted execution. TRUSTJS implements this by evaluating additional attributes of `script` tags: `trustjs-auth` for integrity-protected and `trustjs-encrypt` for confidential code. The following listings illustrate this by an example for two confidential functions. Listing 2 shows two HTML script tags before encryption: while the first one will be executed inside the interpreter enclave, the second one will be executed by the browsers default engine and call the trusted function. The `@exposed` annotation is essential and used by the interpreter enclave for making only developer-defined *exposed functions* callable: Here, `hiddenFunction` is not accessible from untrusted JavaScript. The number in the annotation denotes the number of arguments the function takes.

After encryption, the content looks like shown in Listing 3: The content of the annotated tag is encrypted and embedded as a base64-encoded string, while the calling, untrusted function is untouched. In fact, it will call a proxy function generated by TRUSTJS which matches name and arguments of the original function. Note, that a proxy is only generated for exposed functions and the interpreter enclave prevents calling other functions than these. Additionally, we implemented an HTML generator, which can be used by service providers to generate partly encrypted HTML documents based on these annotations.

```
<script trustjs-encrypt="yes"
  trustjs-blob="X6YXkazAVA7oBZYC..9CkX0Tq9I="/>
<script>var a = confidentialFunction(42);</script>
```

Listing 3: Trusted JavaScript after encryption with encrypted base64-encoded script.

6. EVALUATION

In this section, we demonstrate the benefits that both user and service provider gain from using TRUSTJS, when developers can adapt their approach to save resources. To evaluate the impact of this adoption, we create a setup involving three network entities (client, server and IAS) and explore latency between client and server as well as the scalability of the server software. For the client, we use a typical client machine: a laptop with an i7-6500U CPU. The server is implemented in Node.js and runs on a *t2.medium* Amazon EC2 [3] instance (2 vCPUs, 4 GB RAM), deployed in the *us-east-1a* region. The IAS is a web service operated by Intel. The average ping round trip time between client and server is 99 ms, between server and IAS 25 ms.

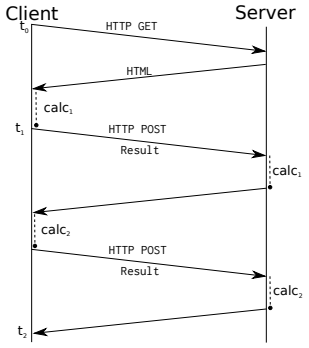
We create a simple web application, that repeatedly performs n calculations $c_1, \ldots, c_n$ on client-side. To show the positive impact of TRUSTJS, we require the results to be checked before giving the client a new task (i.e. c_{i-1} has to be verified before c_i is per-

formed). This requirement allows the TRUSTJS application to do calculations locally and send the combined results to the server. We impose this additional requirement to show that TRUSTJS enables developers to re-think their architectures. In a real world application, this can for example be a multi-step form whose input has to be verified before giving new information to users.

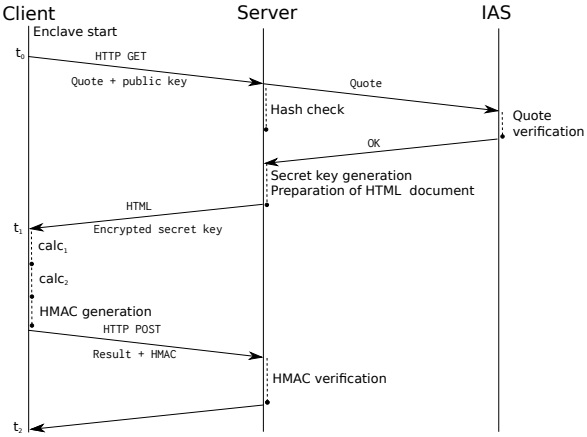
As a place holder for general computations, we use an empty JavaScript loop with 5000 iterations. When not using TRUSTJS, this loop is executed in the default engine of the browser. On contrary, when using TRUSTJS, the loop is executed within the interpreter enclave. In the following, we explore the latency and scalability of applications using TRUSTJS and compare these properties to traditional web applications not relying on TRUSTJS. We do not indicate errors because of negligibility.

Latency. Web applications easily suffer from high latencies, as it is a crucial value, potentially having detrimental effects on user experience. A traditional implementation without TRUSTJS increasingly builds up latency, as can be inferred from Figure 4a. In contrast, as Figure 4b shows, using TRUSTJS can save round trips: The client is trustworthy, thus, multiple server side recalculations can be replaced with one single HMAC verification. Both figures show three points in time t_0, t_1 and t_2 that represent three intervals:

$$I_1 = [t_0, t_1] \qquad I_2 = [t_1, t_2] \qquad I_3 = [t_0, t_2] = I_1 + I_2$$

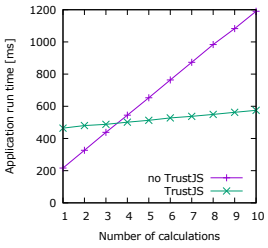


(a) Traditional web application with server-side re-calculations.

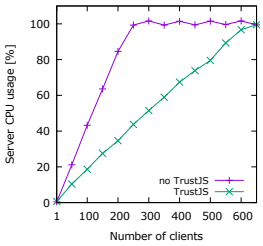


(b) Web application using TrustJS with single server-side HMAC check, SPID and POST URL are assumed to be cached.

Figure 4: Latency build-up of web applications with and without TrustJS, number of calculations $n = 2$.



(a) Run time I_3 of web applications in dependency of the number of calculations n.



(b) Server CPU usage in dependency of client count, number of calculations $n = 4$.

Figure 5: Comparison of latency and scalability properties of web applications with and without TrustJS.

I_1 is the page load time perceived by the user, which is larger for TRUSTJS applications, as it includes a round trip for remote attestation. In contrast, the application run time I_2 depends on the number of calculations performed and thus grows faster for applications not using TRUSTJS. The number of calculations n of our sample application is pivotal for the total application runtime I_3.

With TRUSTJS the average value of I_1 is 343 ms, without it is 109 ms. Figure 5a details the total runtime in dependency of the number of calculations n for both implementations. It shows that our sample application using TRUSTJS experiences shorter latencies compared to a traditional implementation when $n > 3$.

Scalability. In web applications with many users, servers can easily become the bottle neck. We explore the scalability of web servers by observing the CPU utilisation when increasing the number of clients. Figure 5b shows the single core server load for a growing number of simulated clients. It shows, that executing most of the application logic on clients only with TRUSTJS can reduce the server load significantly: Without TRUSTJS, the server reaches full CPU utilisation when serving 250 clients. When relying on TRUSTJS, however, this point is reached at 700 clients.

7. RELATED WORK

Although it was initially assumed that only small tailored applications would be executed inside enclaves [20] such as [10, 25, 7], a recent trend is to consider enclaves as a generic isolation environment for arbitrary applications: VC3 [33] uses enclaves to secure MapReduce jobs; Haven [8] places a library OS inside an enclave for running unmodified Windows applications, Graphene [12] follows a similar approach for legacy Linux applications. SCONE [5] and PANOPLY [34] also enable Linux applications in enclaves, but minimise the TCB by delegating system calls to the host OS.

MiniBox [26] is a general hypervisor-based sandbox for PaaS platforms. Ryoan [21] is an SGX-based distributed sandbox for untrusted platforms such as machine learning services. Both approaches solve a similar problem as TRUSTJS with two-way sandboxing that can run on untrusted systems and can execute code defined by other parties.

Ripley [35] addresses the problem of untrusted clients by automatically replicating the execution of client JavaScript on servers. Although this approach can detect malicious clients, computational resources cannot be saved at server-side. CRYPTON [16] introduces a trusted kernel on client-side providing memory encryption and trusted I/O, but has no capabilities of trusted execution. AutoFBI [39] spawns new browser instances for security relevant web-

sites, which is to some extend a similar approach to our multiple interpreter enclaves. In contrast to TRUSTJS, these solutions do not rely on trusted execution mechanisms as for example provided by SGX and thus can only provide inferior security.

Finally, there are systems that could benefit from TRUSTJS's contributions. Maygh [38] for example is a JavaScript-based content delivery network built on top of untrusted clients. Trust issues are solved by comparing hashes for files being distributed. The trusted execution of JavaScript could help this approach.

8. CONCLUSION AND FUTURE WORK

We have presented TRUSTJS, a novel framework that enables trustworthy execution of client-side JavaScript code inside commodity browsers. TRUSTJS allows service providers to save resources and developers to create better user experiences. We showed that with TRUSTJS, developers can offload computation, resulting in users experiencing lower latencies when they interact frequent enough and servers being able to handle more clients. Still, the extent of saved resources is depended of the use case.

In order to achieve better performance by just-in-time compilation and provide stronger isolation, we plan to use a more sophisticated JavaScript engine for the interpreter enclave, for example V8 [19]. Also, we want to provide richer functionality inside the enclave by supporting parts of the NodeJS API.

9. REFERENCES

[1] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic differential repair for input validation and sanitization. ISSTA, 2014.

[2] Alyson La, GitHub. Language Trends on GitHub. https://github.com/blog/2047-language-trends-on-github, August 2015.

[3] Amazon.com, Inc. Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2, 2017.

[4] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. HASP, 2013.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, et al. SCONE: Secure linux containers with Intel SGX. OSDI, 2016.

[6] Artifex Software, Inc. http://mujs.com/, 2016.

[7] E. Bauman and Z. Lin. A Case for Protecting Computer Games With SGX. SysTEX, 2016.

[8] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. OSDI, 2014.

[9] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. CCS, 2010.

[10] H. Brekalo, R. Strackx, and F. Piessens. Mitigating Password Database Breaches with Intel SGX. SysTEX, 2016.

[11] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. Middleware, 2016.

[12] Chia-Che Tsai. https://github.com/oscarlab/graphene, 2017.

[13] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. USENIX Security, 2011.

[14] CuteSoft. https://javascriptobfuscator.com/, 2016.

[15] X. de Carné de Carnavalet and M. Mannan. Challenges and implications of verifiable builds for security-critical open-source software. ACSAC, 2014.

[16] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting sensitive web content from client-side vulnerabilities with CRYPTONS. CCS, 2013.

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

[18] Google Inc. Understanding web pages better. https://webmasters.googleblog.com/2014/05/understanding-web-pages-better.html, May 2014.

[19] Google Inc. V8. https://developers.google.com/v8/, 2017.

[20] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. HASP, 2013.

[21] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. OSDI, 2016.

[22] Intel Corp. https://01.org/intel-software-guard-extensions, 2016.

[23] JScrambler. https://jscrambler.com/, 2016.

[24] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. SP, 2012.

[25] K. Krawiecka, A. Paverd, and N. Asokan. Protecting Password Databases using Trusted Hardware. SysTEX, 2016.

[26] Y. Li, J. M. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. ATC, 2014.

[27] M. Mikowski and J. Powell. *Single Page Web Applications: JavaScript End-to-end*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.

[28] Mozilla Corporation. Add-on SDK. https://developer.mozilla.org/Add-ons/SDK, 2017.

[29] Mozilla Foundation. js-ctypes. https://developer.mozilla.org/docs/Mozilla/js-ctypes, 2017.

[30] Mozilla Foundation. SpiderMonkey. https://developer.mozilla.org/docs/SpiderMonkey, 2017.

[31] R. Pires, M. Pasin, P. Felber, and C. Fetzer. Secure Content-Based Routing Using Intel Software Guard Extensions. Middleware, 2016.

[32] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. ACSAC, 2010.

[33] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. SP, 2015.

[34] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. NDSS, 2017.

[35] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. CCS, 2009.

[36] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. ESORICS, 2016.

[37] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. SP, 2015.

[38] L. Zhang, F. Zhou, a. Mislove, and R. Sundaram. Maygh: Building a CDN from client web browsers. *EuroSys*, 2013.

[39] M. Zohrevandi and R. A. Bazzi. Auto-FBI: a user-friendly approach for secure access to sensitive content on the web. ACSAC, 2013.