uc3m | Universidad Carlos III de Madrid

Master in Cybersecurity
2022-2023

*Master's Thesis*

# "Secure speedup of the future JavaScript deployments"

Aitor Ruiz Garcia

Director
Pedro Peris López
Madrid, 2023

# Secure speedup of the future JavaScript deployments

Aitor Ruiz Garcia*
*Master in Cybersecurity, Universidad Carlos III de Madrid*

## 1 Abstract

JavaScript has emerged as a fundamental language in global development. Much of the success of Node.js, both in server-side applications and in full-stack web applications (client-server), can be attributed to it.

Node.js was primarily created to enable JavaScript to run on the server. The creator of Node never anticipated that JavaScript would be used for literally everything.

JavaScript is employed in desktop applications using Electron [1], and in mobile applications using React Native [2] or Ionic [3]. As an interpreted language, it has a low entry barrier. As the default language of the web, it also boasts a wealth of interesting libraries.

Node.js took the engine that runs JavaScript on Chrome, known as V8, and wrapped it in additional C++ code to make it possible for it to interact with system calls [4].

All these factors have contributed to the rise of JavaScript as the language of the future. Nonetheless, as the creator himself contends, it has experienced some 'growing pains' [4].

This project is born with the idea of solving some of these problems. The main objective is to create a library that can be used in Deno projects to speed up the most common operations. This library will be written in Rust, a language that is both fast and memory safe.

This operations will be cryptographic operations, such as hashing, encrypting and decrypting. These operations are very common in the web space, and they are very CPU intensive. This is why they are a good candidate for this project.

## 2 The problem

Ryan Dahl, the creator of Node.js, has expressed his regrets about the language [4]. He has stated that he would not use JavaScript for a new project and that he would also not use Node.js for a new project. Node.js has grown beyond what it was intended to be, and it has become a 'monster'.

While experimenting with the TFG [5] project, some problems where experienced with the current state of the JavaScript ecosystem, which they will be explained in the following sections. This TFG project was a web application with blockchain integrations for decentralized identity management. Even though this project is not within the scope of this TFM, it is important to understand the context of the problems that I have discovered.

*Corresponding author.
*E-mail address:100482217@alumnos.uc3m.es*

### 2.1 No types

JavaScript does not have a type checker, at least at *compile* time. JavaScript is an interpreted language, therefore it does not have a compile step. This means that there is no compiler to check the types of the variables, and therefore, it is possible to assign a value of one type to a variable of another type. This can lead to unexpected errors in the code. It is also possible to request a property from an object that does not exist. This will result in a runtime error, with no previous warning.

Take a look at the following example:

```
const user = {
    name: "",
    age: 0
}

console.log(user.email);
```

This small code will not raise any error to warn the developer that the user object does not have an email property. This leads to a runtime error, which is not good for the developer experience.

It is even worse when the user receives the error in production, because the error will be shown to the user, and the user will not understand what is happening.

This happens because JavaScript is a dynamic language and does not have a type checker.

The original TFG [5] employed Next.js, a popular framework for server-rendered React applications. Initially, the challenges that are elaborated in subsequent sections were not foreseen. Next.js was chosen due to familiarity with the framework. However, as the project evolved, issues began to arise, particularly related to the absence of types.

### 2.2 node_modules

Node.js has a package manager called npm. This package manager is used to install libraries in the project. The problem is that the libraries are installed in the `node_modules` folder, and this folder can grow significantly. [6]

By default, npm does not create symbolic links between the libraries. This means that if you have two libraries that use the same dependency, that dependency will be installed twice. More packages installed means more disk space used and more time required to install the packages. A CI build could take a considerable amount of time just to install all the packages.

#### 2.2.1 node_modules security implications

As JavaScript is mainly used in web applications, both on the server and in the client, it is a good target for attacks. Currently, npm holds more than 1.6 million packages. [7]

A successful attack on a web framework could grant access to a large number of browsers, a small percentage of which may be vulnerable.

Although JavaScript in a browser is sandboxed, making it *safer*, this is not the case with Node.js. In Node.js, all the code can interact with the file system and the entire internet.

In the past, there have been multiple cases of compromised npm packages, the most famous being `colors` and `faker.js`. [8] [9]

The developer went rogue and introduced infinite loops, disrupting the normal workflow of multiple developers.

Another notable example was `UAParser.js`, which downloaded and installed a password stealer and a cryptominer. It is important to note that this package is still used by millions of users daily.

### 2.3 Node GYP

Node GYP is the state of the art when creating native libraries in Node JS. This tool is originally from google [4]. When google discontinuated it the comunity made a fork and the project continued. [10]

It is a tool writen in python, witch is not JavaScript, and it is inecessarilly complicated to write a native library.

In order to create a native library, some deep `C++` understanding is needed and the tooling around it is writen in a variety of languagues making it an extra effort to develop some native libraries.

In the original repository they present a simple example of a native library. [10]

It consists of 33 lines of code for a simple Hello world.

```
extern "C" NODE_MODULE_EXPORT void
NODE_MODULE_INITIALIZER(
    v8::Local<v8::Object> exports,
    v8::Local<v8::Value> module,
    v8::Local<v8::Context> context) {
  NODE_SET_METHOD(exports, "hello", Method);
}
```

It needs a lot of boiler plate code just to signal the actual function it is going to be called from JavaScript.

```
static void Method(
    const v8::
    FunctionCallbackInfo<v8::
    Value>&
    args
    ) {
  v8::Isolate* isolate = args.GetIsolate();
  args.GetReturnValue()
      .Set(
          v8::String::NewFromUtf8(
          isolate,
          "world")
          .ToLocalChecked()
    );
}
```

It requires to interact with the V8 engine directly, witch is not a trivial task. This code, even though it is simple, it is not easy to understand. It just returns a string with the value `world`.

This code will be more complicated in some other C ABI compatible languages, because they would need to implement the V8 engine in their language, to then be able to interact with it.

Rust, that will be discused later has a crate that links directly with V8, but it is not a trivial task to implement.

## 3 THE SOLUTION

Ryan Dahl, created Deno, a solution to his own issues created by working on Node JS without a clear plan. In a talk at JSConf [4] he presented the project with some clear plan to make Deno the standard langua for the web.

His plan was to create a safe and secure environment for JavaScript and TypeScript to run. This means that the developer will not have to worry about security issues, as the code will be running in a sandboxed environment.

By default Deno should not allow access to the file system or the internet. As JavaScript by nature is a sandboxed environment this step is just natural to take.

Deno should works via message passing between the host and the sandboxed environment. This means that the host will not have direct access to the sandboxed environment. This is a very good security measure, as the host will not be able to access the sandboxed environment. It makes embeding Deno in other projects very easy, as the developer must only implement the message passing protocol.

The whole experience would be almost the same as in the web. In the web the imports works via URL, and Deno should work the same way.

Here are some of the solutions that were developed.

### 3.1 TypeScript

TypeScript appeared in 2012 developed by Microsoft, to add types to JavaScript. It works as a superset of JavaScript, meaning all keywords valid in JavaScript are also valid in TypeScript, but not vice versa. [11] [12]

TypeScript does not work by modifying V8; instead, it makes JavaScript the compile target of TypeScript. This approach also allows for targeting multiple versions of JavaScript and the use of polyfills [13] if the desired API is not present.

However, while TypeScript in this form is very helpful, it adds more time to the development process. This is because it first has to compile TypeScript to JavaScript and then run the intended program. This is the case in most of Node.js projects. In most cases the `package.json` can appear as follows:

```
"scripts": {
    "start": "node index.js",
    "build": "tsc"
}
```

This means that the developer has to run the `build` script before running the `start` script. This is not a problem in development, but it is a problem in production. This is because the developer has to run the `build` script before deploying the application.

Deno translates TypeScript on the fly and feeds it to V8, meaning the whole process is plug-and-play. Developers do not have to configure an external tool, as `tsc`, the compiler, can be difficult to set up and configure.

In the code, this means that every variable will have an associated type, and they cannot change types anywhere in the code.

```
var name: string = "";
```

The variable name cannot have another value assigned without it being a string.

```
name = 0;
```

This code will result in an error, as `0` is not a string.

#### 3.1.1 JSDoc

While TypeScript is a very good solution, it can be a pain to set up. It requires a lot of configuration, and it can be difficult to set up in some projects. While JSDoc main focus is to provide documentation abount functions to let developers know how a function can be used, it can also be used to provide types to variables.

It is even possible to generate TypeScript types from JSDoc comments.

```
/**
 * @type {string}
 */
var name = "";
```

Just like that a variable can have a type. With this information a `.d.ts` file can be generated and tsc will read it to apply type checking to native JavaScript code.

Some projects stand out in this space like Svelte and Turbo.

#### 3.1.2 Svelte

Svelte [14], is a framework that creates UIs, just like React. It is a very good alternative to React, as it is faster and has a smaller bundle size.

In an article, the Svelte author discusses the benefits of using TypeScript in Svelte. In his oppinion, TypeScript is *"not worth it"* for creating a framework.

He argues that TypeScript is very good but in his case, the hustle of setting up TypeScript is too much for the benefits it provides.

Most of their troubles comes from generics and types magic that is required in some edge cases on their framework.

*"My position is, types are fantastic, TypeScript is a bit of a pain ... as soon as you use a .ts file, then you have to have the tooling to support that ... there's all of these points of friction when you use a non-standard language like TypeScript that I have come to believe makes it not worth it. So instead, we have put all our types in JSDoc annotations, and we get all of the type safety, but none of the drawbacks, because it is just JavaScript, everything is in comments, you can just run the code. This is what we do in the Sveltekit codebase and it has worked out fantastically so for Svelte 4.0, we're going to do the same for Svelte because it's going to enable us to move much more quickly." - Rich Harris* [15]

#### 3.1.3 Turbo

Turbo, is a library that aims to make the life of backend developer easier, allowing the insertion of HTML in a page directly without a full page reload.

Very recently, the lead developer of the library and the creator of Ruby on Rails, anounced that they are droping TypeScript entirely from the project. This also means that any other project that depends on Turbo will also have to adapt as they do not plan on giving any type information.

His main argument is that types damage the code readability and they are not worth the hustle.

*"TypeScript just gets in the way of that for me. Not just because it requires an explicit compile step, but because it pollutes the code with type gymnastics that add ever so little joy to my development experience, and quite frequently considerable grief. Things that should be easy become hard, and things that are hard become 'any'. No thanks!" - David Heinemeier Hansson* [16]

#### 3.1.4 ECMAScript proposal

There is currently an ECMAScript proposal in the making that aims to bring types to native JavaScript. It is still in stage 1, and it will be in that stage for a long time, but it is very promising.

The main benefit is no more build steps. The developer will have types directly in JavaScript. This will unlock the following from the language.

- Less time spent on build steps.
- Cleaner code than JSDoc.
- Engine maker will be able to optimize the code better.

But in the meantime, how can the developer experience be imporved?

#### 3.1.5 How Deno can imporove the TypeScript experience

Using Deno there are no more build steps. This means that the developer can write TypeScript and run it directly in Deno. This is a very good developer experience, as the developer does not have to worry about the build step.

Wanting no types in your code, can also be valid, so Deno allows the execution of JavaScript also. Allowing the best of both worlds.

When the scope of the project is not well defined and a quick iteration process is wanted, JavaScript can be used, and then when the idea mature, TypeScript can be implemented.

Most of this library authors problem come from third party libraries that that are messy to work with, but Deno can and will make the experience smother.

### 3.2 URL based imports

A major change with Deno comes with the URL-based imports; the modules are cached and reused when needed.

However, with this approach, since the script comes from a URL, it may contain malicious code. How could developers protect themselves?

#### 3.2.1 Security checks

Deno implements multiple checks to prevent uncontrolled access to the machine. [17]

- **–allow-env** Allows the reading of `.env` files.
- **–allow-sys** Allows the reading of system-specific information, such as the OS version or information about memory.
- **–allow-hrtime** Allows high-resolution time measurement. High-resolution time can be used in timing attacks and fingerprinting.
- **–allow-net** Allows network access. You can specify an optional, comma-separated list of IP addresses or hostnames (optionally with ports) to provide an allow-list of permitted network addresses.
- **–allow-ffi** Allows the loading of dynamic libraries. This is still an unstable feature.
- **–allow-read** Allows file system read access.
- **–allow-run** Allows the running of subprocesses.
- **–allow-write** Allows file system write access.

All these flags have their opposite counterparts and can be used to allow access to certain folders while disallowing specific files. For example:

```
--allow-read=/etc \
--deny-read=/etc/hosts
```

### 3.3 Speedup

The solution that some project owners have implemented to improve performance is to use a compiled language as a companion. These languages *in general* have better performance than interpreted ones.

In the web space, two projects stand out from the rest. The developer ecosystem around web development has grown exponentially, and now, more than ever, a bundler [18] is needed to combine all that information into a simple combination of HTML, CSS, and JS.

`SWC` and `esbuild` are both tools that rely on other languages for speedup. However, these two projects played a significant role in the development of this project.

#### 3.3.1 SWC

SWC (Speedy Web Compiler) is a project that aims to speed up the compilation of JavaScript and TypeScript. This means it will convert an existing TypeScript codebase and convert it to JavaScript, while keeping everything tidy and compact.

As not all browsers support the latest JavaScript features, SWC can also convert the code to an older version of JavaScript. There is a whole list of browsers that SWC supports, and it is quite impressive.

SWC decided to compile a Rust program and turn it into a binary. Later, wrote some JS glue code to make it usable in a JS project.

As this application is a pure CLI-style application, this approach is beneficial as it only requires a small amount of JS code. This is because they only need to pass the necessary arguments to the Rust binary, and then the Rust program takes over and packages the JS application.

While this can be beneficial for this objective, for a library that needs to run when the developer calls a function, the glue code must spawn a shell, and that takes time. So much time, in fact, that it may not even be beneficial to use a native library.

#### 3.3.2 esbuild

esbuild is a bundler [18] that aims to be the fastest bundler in the world. A bundler combines all dependencies into a single file to distrite it to the users. This is a very common practice in the web space, as it can also split the code into chunks to be loaded on demand. This is very useful for large applications, as it can reduce the initial load time. This is the case of the TFG [5] project, as it is a large application.

`esbuild` decided to take another approach. They used wasm [19] [20] to create the library. As this project is written in Go, the whole Go garbage collector has to be injected into the wasm code.

This can result in larger install sizes and a poor developer experience, as stated before.

It's worth noting that wasm, even though it is excellent technology and has brought significant speedup to the web, still lacks a good memory management solution. This results in very good performance compared to JavaScript, but not compared to other technologies.

## 4 THE CODE

The arrived solution is a dynamic library with minimal TypeScript glue code.

Deno offers incredible dynamic library support with their native API.

```
Deno.dlopen(
    test.lib,
    {},
)
```

Deno supports the whole C ABI, meaning that it can load any dynamic library that is compatible with the C ABI. This is very good, as it allows the developer to use any language that is compatible with the C ABI.

| FFI Type | Deno | Rust |
|---|---|---|
| i8 | number | i8 |
| u8 | number | u8 |
| i16 | number | i16 |
| u16 | number | u16 |
| i32 | number | i32 |
| u32 | number | u32 |
| i64 | number — bigint | i64 |
| u64 | number — bigint | u64 |
| usize | number — bigint | usize |
| f32 | number — bigint | f32 |
| f64 | number — bigint | f64 |
| void[1] | undefined | () |
| pointer | {} — null | *mut c_void |
| buffer[2] | TypedArray — null | *mut u8 |
| function[3] | {} — null | Option< extern "C" fn() > |
| { struct: [...] }[4] | TypedArray | MyStruct |

As this project only uses strings and numbers, the only times needed are a pointer and a u32. So the type declaration of the dynamic library is as follows.

```
hash: {
    parameters: ["pointer", "u32"],
    result: "pointer",
    nonblocking: true,
}
```

This is the declaration of the hash function in the library. In the declaration it can be seen a `nonblocking` flag. This flag is used to tell Deno that the function is async. Instead of complicated C ABI callbacks Deno converts the sync Rust code in async JavaScript code automatically.

All of the functions in the library are async because in the Deno ecosystem async functions are easier to work with. This is because Deno has top level await and it is not necessary to wrap the code in promises or callbacks.

This is the table that Deno uses to convert the types from JavaScript to Rust. It has some Rust magic to convert the types from C ABI to Rust types. It allows the convertion of `unsafe` operations to safe operations while having an excellent developer experience and performance.

In this method, the developer simply has to locate the DLL physically. The developer also needs to specify how the methods in the library work. The process would be as follows:

1) Check the file system to see if the DLL is present. If the user did not allow this from the launch flags, Deno will ask for permission.
2) Download the DLL if it is not present. If the user did not allow this from the launch flags, Deno will ask for permission.
3) Load the library and prepare it.

### 4.1 Speedy but memory safe

The final piece of this project is deciding how to secure the actual horsepower of this project. The obvious choice is a low-level language that can maximize the performance of the CPU.

Here is a list of compiled languages that may be a good fit for this project.

- C
- C++
- Rust
- Go
- Zig

To prevent memory-related issues, some languages have to be ruled out. In languages like C and C++, memory is managed by the developer, which can result in segmentation faults and other issues. [21]

A garbage collector will not be a good fit for this project, as it can result in performance issues. It also can result in big library sizes, as the garbage collector has to be included in the library.

Zig, while being a very good language, is still in its early stages, and it is not a good fit for this project. It also does not have a safe memory management solution, so it will not be considered as an option much like C. Although, it is worth noting that Zig is a very good language, and it is worth keeping an eye on it. The `defer` keyword is a very good feature that can be used to keep track of memory allocation.

The only sensible option left is `Rust`.

Rust uses a borrow checker to keep track of all allocated memory while not allowing the user to allocate and deallocate manually. This way, it can guarantee that all the needed memory will be used efficiently.

If a variable is borrowed, it means that it originates from another function, and the current function cannot take ownership of the variable. The variable will continue to exist.

If a variable is owned, it will be deallocated when it goes out of scope.

```
const a = 10;
fun(a);
foo(a);
```

In this code, only the first function call will comply with the compiler. The other one will fail.

To make this code work, it will need to be changed.

```
const a = 10;
fun(&a);
foo(a);
```

As the first function does not take ownership of the variable `a`, it is not deallocated at the end and can be used in the subsequent function call.

However there is limitation to the borrow checker. It only works if the compailer knows for certain that the variable will exists and the memory will be allocated. Unfortunatelly is not the case when Rust has to interacts with the outside world. I/O operations are not guaranteed to be successful, and therefore, the borrow checker cannot be used. This is the case of this project, as it has to interact with Deno. Rust has to *trust* Deno to allocate and pass that memory to the Rust library.

Rust has a *scary* keyword named `unsafe`. It disables the borrow checker and allows the developer to do *whatever they want* memory-wise.

To convert an array of u8 bytes, witch is in fact an string, to a Rust string, the Rust language includes some methods to make this process as safe as it can.

```
let password: &str;

unsafe {
    password =
        CStr::from_ptr(input_text_pointer)
        .to_str()
        .expect("Could not parse string.");
}
```

Rust has a method to convert a pointer to a string, but it is not safe. It can fail, and therefore, it is wrapped in a `Result` type. This type can be either `Ok` or `Err`. If it is `Ok`, the string is returned, and if it is `Err`, the program will panic. This panic will have to be resolved in Deno. A panic will result in a null being returned from the FFI bridge. Checking for null will be in the necessary glue code of this library.

#### 4.1.1 Creating a dynamic library

In Rust, everything is just a compilation target, so it can simply be specified that a library is desired. This can be achived by adding the following to the `Cargo.toml` file.

```
[lib]
crate-type = ["cdylib"]
```

This tells `rustc` to create a `.dll` in windows or a `.so` in linux. This is the only thing that needs to be done to create a dynamic library. The rest is just Rust code.

To make it fully usable in Deno, the Rust code needs to be wrapped in a `extern "C"` block. This will make the Rust code callable from C, and therefore, from Deno. This is done as follows:

```
#[no_mangle]
pub extern "C" fn hi() {
    // Code goes here
}
```

There are multiple types of libraries, but the focus of this project is to create a library that can be used in other projects. It can be used in other project because it is a dynamic library, and it is not a static library. A static library needs to be compiled into the final binary, and can only be used inside that program. A dynamic library can exist in a separate file and can be used in multiple programs. This is the case of this project.

A fear when using FFI is that it may result in a performance penalty. However, taking a flamegraph [1] shows that that is not the case. The FFI bridge is not even visible in the flamegraph. It is just a transparent step. It does not add any overhead to the execution of the program. Between the `deno.exe ffi_call_win64()` and the actual `tfm.dll tfm::verify` function starting execution there is no time difference.

### 4.2 Where WASM fits

WASM, as mentioned before, is quite useful in the browser, where it belongs. Right now, it is still in its early stages, and the WASM API within the browser is still young. The WASI libc library works, but it is not fully matured yet.

This is cobered in a Github issue. [22]

- No way to control in a guaranteed fashion when new memory commit vs address space reserve occurs.
- No way to uncommit used memory pages.
- No way to shrink the allocated Wasm Memory.
- No virtual memory support (leading applications to either expect to always be able to grow, or have to implement memory defrag solutions)
- If Memory is Shared, then application needs to know the Maximum memory size ahead of time, or gratuitously reserve all that it can.

The author summarizes the issue by saying, *you can only grab more memory, with no guarantee if the memory you got is a reserve or a commit*.

In WASM there is not way to allocate memory in compile time, witch would be benefitial for this project. WASM not being able to provide this functionality is a big limitation for this project, and results in a poorer performance.

### 4.3 The data

Let's take a look at the actual data to see if this idea actually holds up. These tests have been conducted using strings from a collection of 36 characters repeated 1,000 times to the same collection of 36 characters repeated 2,000 times. Each set is repeated 15 times, and the average is calculated to rule out any errors.

The data is collected sequentially, meaning that no async functions are used and the Deno and Rust functions are run one after the other to rule out any OS-induced noise.

#### 4.3.1 Hardware used

These tests have been run on a broad range of hardware, all of the `x86_64` architecture.

- GitHub Actions builders 2-core CPU `x86_64` [23]
- i9-12900H
- i5-1135G7

The data presented in this document will come from the CPU **i9-12900H**. However, there is data available in the GitHub repository that can be extracted from every commit added to the repo.

In every test run, the results are consistent but with obvious speed differences, as the selected hardware varies widely.

The **i9-12900H** is a 16-core CPU with 24 threads. It has a base clock of 2.2 GHz and a boost clock of 4.9 GHz. It has 24 MB of L3 cache and 16 MB of L2 cache. It is a very powerful CPU, and has a powerful single core performance. This means that it will go in favor of Deno. As reducing the speed of the processor will result in higher time values for Deno.

#### 4.3.2 Hashing

The first part of the library to analyze is hashing. The algorithm used for hashing is bcrypt. This algorithm is a slow hashing algorithm that is designed to hash passwords. It is intentionally slow to prevent brute-force attacks [24]. The algorithm can be configured to use a different "cost," which is the number of iterations that the algorithm will perform. The higher the cost, the more secure the hash will be, but this comes at the expense of time. It also includes a salt to prevent rainbow table attacks.

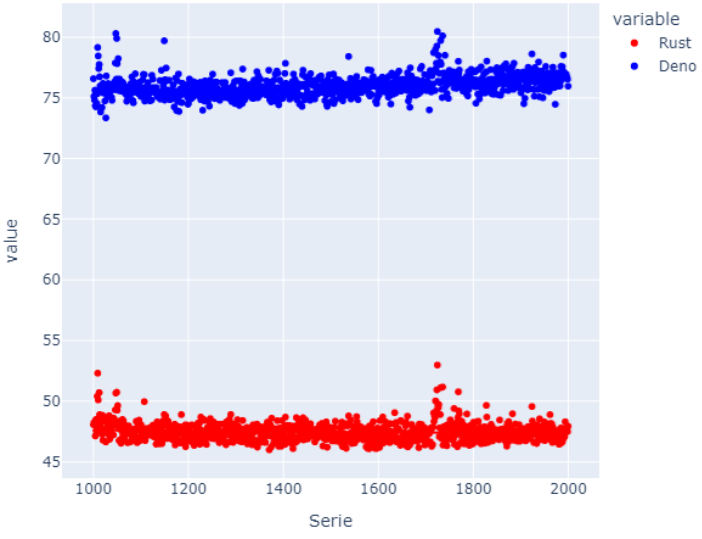
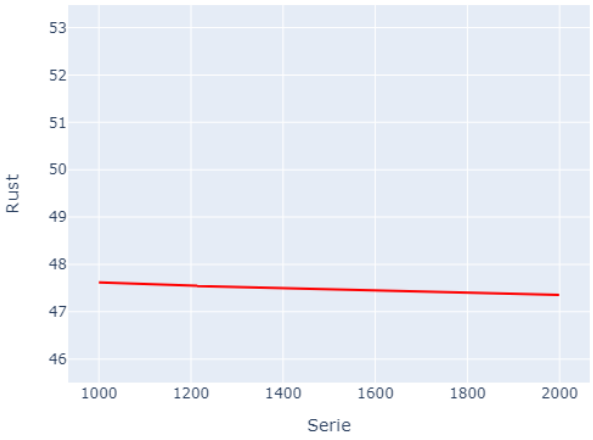



Figure 1: Flamegraph of the verify function.

In this image, the creation of 1,000 hashes and their respective costs in milliseconds are shown, using a bcrypt cost of 14. On average, Rust is faster by **57%**. Reducing the hardware choice to **i5-1135G7** results in a **34%** more slowdown of Deno compared to Rust. Reducing even more the hardware to a single core instance of a VM runing on DigitalOcean, the results are even more drastic, with a **58%** slowdown of Deno compared to Rust. This results are consistants with the rest of tests runned.

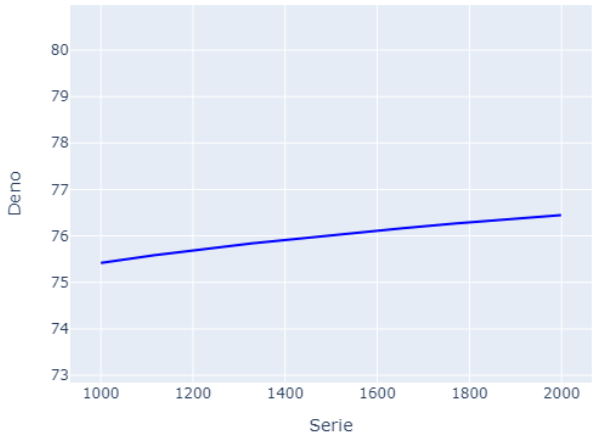



In this hash graph a small OS induced noise can be seen, and ruled out as Deno or Rust fault. At the start of the series and between 1600 and 1800 can be detected. As both timings go up it is safe to rule out this data pasing to the next section. The trends here are not clearly defined, but two aspects are apparent from the following graphs.

From the data, some observations can be made:

- The cost of hashing will increase more rapidly than it does in Rust.
- Rust will be more stable and experience less growth.

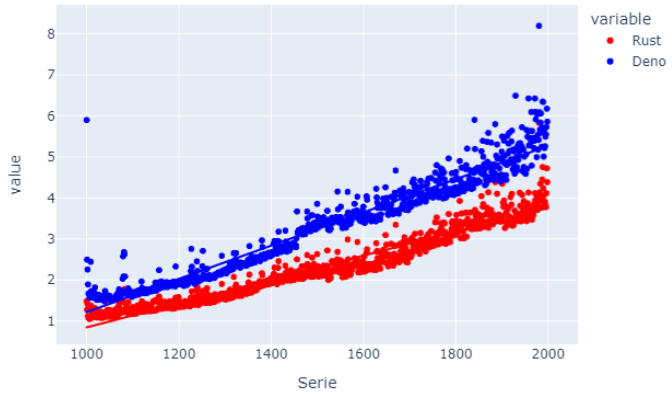
The growth rate in the Deno is **0.001** and in Rust is in the range of the data chosen negible. Pushing it all the way to **1.800.000** characters will result in a growth rate of **0.0001**.

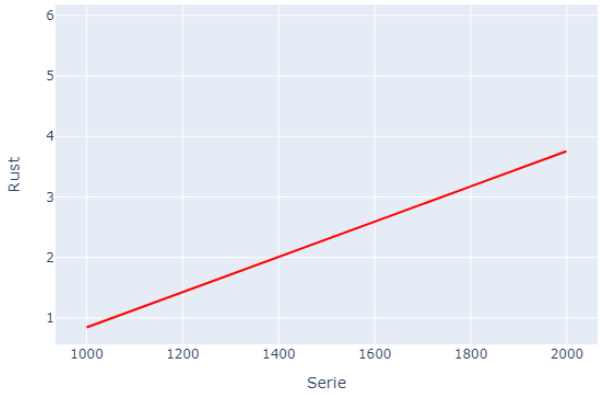



### *4.3.3 Secret boxes*

Secret boxes are a concept introduced by the library NaCl. A secret box is essentially a box with only one lock, serving as a metaphor for synchronous cryptography. The algorithm to be used in this project is a variant called *tweetnacl*, which replicates the exact functionality of NaCl but fits within 100 tweets.

The focus of this library is on speed and minimal disk footprint for both developers and users.




From the data, some observations can be made:

- The cost of encrypting will rise more rapidly in JavaScript than it does in Rust.
- Rust will be more stable and experience less growth.

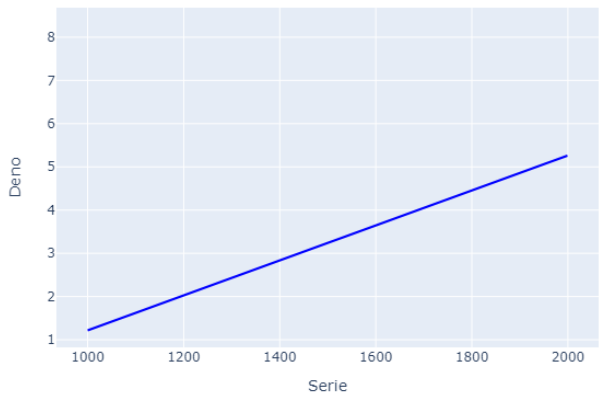
The growth rate in the Deno is **0.004** and in Rust is **0.003**.

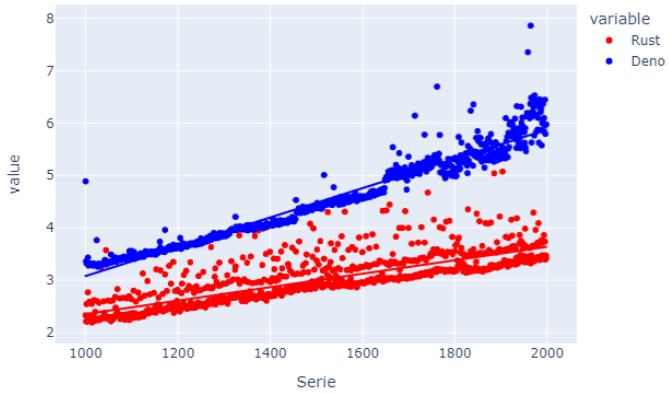
In the graph, there are some Deno results that deviate from the predicted outcomes. Since these anomalies do not appear in the Rust variant, they likely arise from variations in V8's *heat* function. In V8, when a function is called, it gains *heat*, meaning it gets cached for later use. If a function is executed frequently, it gets stored in machine code to maximize performance.

In this example, it can once again be observed that JavaScript is more unstable than Rust and tends to experience greater growth in computational cost.

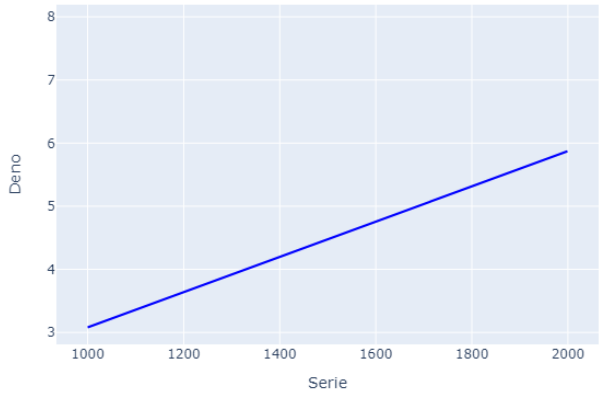
#### 4.3.4 Box

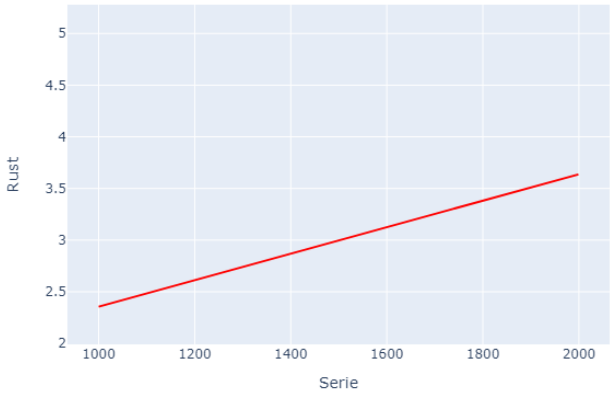
A box is essentially a container with two locks, serving as another metaphor for asymmetric encryption. Like the previous algorithm, this one is based on *tweetnacl*.




In this final algorithm, Rust outperforms Deno significantly, even though its performance is more dispersed.




From the data, some aspects can be deduced:

- The cost of encrypting will rise more rapidly in Deno than in Rust.
- Rust will be more stable and experience less growth.

The growth rate in the Deno is **0.003** and in Rust is **0.001**.

## 5 Conclusions

The objective of this project has been met with remarkable success, both theoretically and pragmatically. The initiative was principally driven by the need to enhance the efficiency of JavaScript-based projects, a technology that is increasingly becoming ubiquitous. One notable application of this effort is seen in the InterPlanetary File System (IPFS) [25], which heavily relies on our optimized suite of algorithms for its operations. Particularly, this suite of libraries plays a pivotal role in the functionality of OrbitDB, an instrumental library employed for the creation of decentralized databases.

### 5.1 Technical Overview

The libraries developed under this project are predicated on distributed technology paradigms. These paradigms operate on the principle of data transmission involving multiple peers, wherein each peer sends fractional pieces of information to a centralized entity. Given this architecture, the need for high-speed data aggregation and dissemination is critical. Such requirements present unique challenges when relying on traditional methods of information transport.

### 5.2 Performance Limitations in IPFS

Although IPFS is fundamentally a JavaScript-based project, it is not immune to the performance bottlenecks commonly associated with the language. Despite commendable optimization efforts by the IPFS development team, the potential of the system is constricted by the inherent limitations of JavaScript.

A considerable breakthrough was achieved when IPFS was implemented in the Go programming language, known for its C Application Binary Interface (ABI) and robust WebAssembly (WASM) capabilities. Nevertheless, the core functionalities still rely on JavaScript. While multiple implementations of the IPFS protocol exist, the official version remains the most extensively deployed, resulting in significant network latency.

### 5.3 Performance Augmentation

The integration of our library promises to drastically improve performance metrics, both in desktop applications and browser-based environments.

### 5.4 Future-Proofing Developer Environments

In this project, it is made enphasis in the use of Deno because of its sandboxed environment. The next logical step to prevent malicious code from being executed in developers machines is the use of containers. This way, the developer can be sure that the code is not doing anything malicious. Through the utilization of Docker containers. This virtualization allows for a consistent and reproducible development atmosphere, replete with all the necessary dependencies. It also serves as a security measure, as the operational codebase resides within a contained environment, effectively neutralizing potential threats.

### 5.5 Industry Leadership

Microsoft is at the forefront of this technological space, particularly with their innovative feature in Visual Studio Code known as Remote Containers. This utility allows developers to execute code within a containerized environment directly from a specific folder, obviating the need for local dependency installations. This feature is not just advantageous for enhancing the developer experience but also critically important for bolstering security protocols.

Microsoft offers a local version of this feature, named `dev-containers`, that provides identical functionalities but operates within a locally-hosted environment.

## References

[1] Build cross-platform desktop apps with javascript, html, and css. https://www.electronjs.org/, Accessed: 08/09/2023.
[2] React native learn once, write anywhere. https://reactnative.dev/, Accessed: 08/09/2023.

[3] The mobile sdk for the web. https://ionicframework.com/, Accessed: 08/09/2023.

[4] Ryan Dahl. 10 things i regret about node.js. In *JSConf EU 20202*, 2022.

[5] Aitor Ruiz Garcia. Investigación y desarrollo del uso de la tecnoloía blockchain como validador de indentidades distribuidas y su aplicación en la universidad de deusto. Master's thesis, Universidad de Deusto, 2022.

[6] Mateusz Morszczyzna. What's really wrong with node_modules and why this is your fault. https://hackernoon.com/whats-really-wrong-with-node-modules-and-why-this-is-your-fault-8ac9fa893823,, 2017.

[7] Npm. https://www.wikiwand.com/en/Npm, Accessed: 08/09/2023.

[8] Ax Sharma. npm libraries 'colors' and 'faker' sabotaged in protest by their maintainer—what to do now? https://blog.sonatype.com/npm-libraries-colors-and-faker-sabotaged-in-protest-by-their-maintainer-what-to-do-now,, 2022.

[9] Emma Roth. Open source developer corrupts widely-used libraries, affecting tons of projects. https://www.theverge.com/2022/1/9/22874949/developer-corrupts-open-source-libraries-projects-affected,, 2022.

[10] Node.js native addon build tool. https://github.com/nodejs/node-gyp, 2023.

[11] Typescript. https://www.wikiwand.com/en/TypeScript, Accessed: 08/09/2023.

[12] Ben Grynhaus, Jordan Hudgens, Rayon Hunte, Matt Morgan, and Vekoslav Stefanovski. 2021.

[13] Mdn web docs glossary: Definitions of web-related terms - polyfill. https://developer.mozilla.org/en-US/docs/Glossary/Polyfill, Accessed: 08/09/2023.

[14] Saumyamani Bhardwaz and Rohan Godha. Svelte.js: The most loved framework today. In *2023 2nd International Conference for Innovation in Technology (INOCON)*, pages 1–7, 2023.

[15] Tim Anderson. Typescript is 'not worth it' for developing libraries, says svelte author, as team switches to javascript and jsdoc. *Dev Class*, 2023.

[16] David Heinemeier Hansson. Turbo 8 is dropping typescript. 2023.

[17] Deno - permissions. https://deno.land/manual@v1.36.4/basics/permissions,, Accessed: 08/09/2023.

[18] Diego Salinas Gardón. The complete javascript module bundlers guide. https://snipcart.com/blog/javascript-module-bundler,, Accessed: 08/09/2023.

[19] Mozilla. Mdn web docs - webassembly. https://developer.mozilla.org/en-US/docs/WebAssembly,, Accessed: 08/09/2023.

[20] WebAssembly Community Group. *WebAssembly Specification*. 2023.

[21] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. In rust we trust – a transpiler from unsafe c to safer rust. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 354–355, 2022.

[22] juj. Wasm needs a better memory management story. https://github.com/WebAssembly/design/issues/1397,, 2021.

[23] Github. About github-hosted runners. https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners,, 2023.

[24] Bcrypt. https://www.wikiwand.com/en/Bcrypt, Accessed: 08/09/2023.

[25] Gareth Tyson . . . Dennis Trautwein, Aravindh Raman. Design and evaluation of ipfs: A storage layer for the decentralized web. 2022.