SimpliC in the browser

Wilmer Nilsson
D15, Lund University, Sweden
dat15wni@student.lu.se

Andreas Trattner
D15, Lund University, Sweden
eng11atr@student.lu.se

Abstract

More and more software has been making its way into the browser. Usually this involves rewriting the software in web specific languages, or creating complex APIs with which the front-end can communicate with a high-performance back-end. Now, however, with the help of WebAssembly, a general-purpose virtual instruction set created in the likeness of low-level assembly languages, it is possible to weave performant code into JavaScript, a typical front-end language, which is run through the V8 engine, potentially increasing performance.

At the moment, WebAssembly is able to be generated from C, C++, Rust, Go and all LLVM compatible languages. In this paper, the potential advantages and disadvantages to this approach of web programming will be discussed. The creation of a WebAssembly-compiler will also be explored, using a simplified C-like source language.

With our compiler ready we then perform several benchmarks, comparing the speed of emcc (C to WebAssembly) compiled WebAssembly code to, amongst others, our own compiler's and Javascript's. As expected our compiler was the slowest contester, finishing just after Javascript. The emcc-generated WebAssembly code however, is faster than Javascript albeit slower than languages such as Rust and Java, when run natively. With JIT-compilation turned off in Javascript even our compiler produces code that is magnitudes faster–a fact that could be used to emphasise the need of JIT-functionality in WebAssembly.

1 Introduction

Web applications are rather flexible programs; the client may interact with a program or service, such as Netflix, through the front-end in the web browser. The back-end, on the other hand, may either be locally- or remotely accessed, performing the heavier calculations. Following the Netflix example, the back-end might be the server providing the video feed, transcoding (i.e. compressing or encoding in real-time) the stream if needed. This makes web programs highly portable in nature, as they may be run on any device, as long as they supply a modern web browser.

This flexibility has extended to desktop applications, such as balenaEtcher (a program used to write image files onto storage media), which runs JavaScript locally through the Node.js-framework.

Course paper, EDAN70, Lund University, Sweden
January 18, 2020.

A drawback of this approach is the design of the back-end; In some cases, it's desirable to write both the front-end and back-end in JavaScript, especially if the back-end is executed on the client-side. This makes the bridge between front-end and back-end seamless. JavaScript is, however, JIT-compiled, which might affect the performance negatively. In many cases this is not an issue however, as the accepted performance depends on the application.

Simultaneously, constructing a back-end in a compiled, optimised language introduces a new issue - how does the front-end communicate with the back-end? One solution is introducing some bridge mechanism, such as a Remote Procedure Call-protocol. A RPC protocol allow the caller (client) to send procedure requests to the server through some portable format, and receives a response when the operation has completed on the server side[88].

WebAssembly allows web applications to take the best properties from both of these approaches. It runs natively in the JavaScript engine. At the same time, it is optimised beforehand.

By writing procedures which may be computationally heavy in a high-performance, compile-time optimised language such as C, C++ or Rust, these procedures may be compiled into WebAssembly and imported into- and called from the JavaScript runtime directly, as if they were JavaScript functions.

2 WebAssembly

WebAssembly is an intermediate language, similar in appearance to x86-assembly, and in functionality to Java bytecode.

It is based on the concept of a stack machine; all WebAssembly code does is push, pop- or operate on values on a virtual, implicit stack. The user does not have to explicitly rearrange stack pointers, return addresses etc.[Ros+nd]

The only four types in the language are **i32**, **i64**, **f32** and **f64**. These represents signed integers and floats of 32 and 64 bytes respectively. It is completely platform-independent, and also web-agnostic; the specification does not mention the web in any measure[Pro+19].

Its conception was driven mainly by the desire to improve performance in web applications. In contrast to JavaScript, WebAssembly does not have a garbage collector, which can impact performance unpredictably[Pro+19]. Another argument for its creation is to counteract certain potential security flaws in JavaScript. Historically, many large web hosts such as Facebook, Yahoo and Google have used intricate sandboxing methods in order to encapsulate JavaScript code from untrusted sources. this means that the untrusted code could only be interacted with through certain API:s while limiting or preventing the usage of "unsafe" operations and rebindings[Tal+11]. WebAssembly's semantics and static typing migitates this, as it is easier to analyse and reason about[Pro+19]. This report is, however, more focused on the performance aspect of WebAssembly.

Certain high-level programming languages may be compiled into WebAssembly, such as C, C++ and Rust. WebAssembly exists in two different formats:

1. .wasm: Portable binary code format
2. .wat: Human-readable WebAssembly, written as S-expressions.

Although the .wasm-format is needed to run WebAssembly, a programmer may seamlessly convert files between these formats through the use of wat2wasm and wasm2wat, supplied by the WebAssembly Binary Toolkit [1]. This process is analogous to converting files between Machine language and Assembly Language.

As previously mentioned, the .wat-format is written using S-expressions. This is not required however, as a linear format works identically:

```
i32.const 1
i32.const 3
i32.add
set_local $variable
```

```
(set_local $variable (i32.add (i32.const 1) (i32.const 3)))
```

Both of the code snippets above sets the value of a local variable called "variable" to the value of 1 plus 3.

The uppermost snippet shows how WebAssembly operates on the virtual stack: the two values 1 and 3 are pushed on top of the stack. The add operator is then called, popping the two uppermost values, adding them together and pushing the result back on the stack. Finally, the result is popped from the stack, and stored in the local variable.

S-expressions, first popularized with the programming language Lisp, uses a variant of the function notation $f(e_1, ..., e_n)$ called sequence notation $(f, e_1, ..., e_n)$, for some function f and parameters $e_1, ...e_n$.[Dai15]

2.1 WebAssembly implementation

Most JavaScript engines, such as V8 (Chrome) and Spider-Monkey (Firefox), use their JavaScript compilers to create optimized native binaries from WebAssembly modules ahead of time. Chakra (Edge) however, performs lazy translation to an internal bytecode format upon execution of the WebAssembly module, performing JIT-compilations on repeatedly used code[Ros+nd].

Furthermore, WebAssembly is designed to allow *streaming*; an engine may compile individual functions and use them before the entire binary has been loaded completely[Ros+nd].

[1]https://github.com/WebAssembly/wabt

3 SimpliC

SimpliC is a language which consists of a limited subset of the C language, and as such its usefulness outside of learning about compilers is also limited. It is used to teach compiler implementation in the course EDAN65 at the Faculty of Engineering at Lund's University [2].

The SimpliC language is capable of:

1. Function calls
2. Local variables
3. Integer arithmetic
4. Integer comparison
5. Looping
6. Conditional execution
7. Basic I/O operations

3.1 Data types

There is only one data type in SimpliC: Integers. As such, all functions are required to return an integer value, and may or may not take integers as parameters (Although the language may evaluate booleans in certain cases, see 3.5) Integers may be used as hard-coded constants, in e.g. function calls, or local variables. SimpliC has no notion of static or global variables.

All data in SimpliC uses stack-dynamic allocation.

3.2 I/O

The I/O operations are limited to the two built-in functions "print" and "read", printing single whole numbers to standard output, and reading integers from standard input respectively.

3.3 Arithmetic operators

SimpliC allows addition, subtraction, multiplication, division (quotient of the division between integers), and modulo-division (remainder of the division between integers), and is performed with the infix operators +, -, *, / and % respectively.

3.4 Boolean operators

Integers may be compared by Less-Than,Less-or-Equal-To, Equal, Not-Equal, Greater-Than and Greater-or-Equal-To with the infix operators <, <=, ==, !=, >, >= respectively.

3.5 Control flow

SimpliC has support for C-style While-statements, If-statements, and If-Else-statements. In these control flow statements, single conditionals in the shape of integer comparisons may be evaluated implicitly to booleans, although the language itself has no support for the boolean data type, and multiple separate integer comparisons may not be joined with the "AND"-operator or the "OR"-operator.

[2]http://fileadmin.cs.lth.se/cs/Education/EDAN65/2019/web/index.html

4 Our implementation

Using JastAdd[3], a meta-compilation system, our SimpliC to WebAssembly compiler manages to compile a SimpliC program into the WebAssembly text format. In order for the generated file to be useful in JavaScript, it has to be converted using the tool "wat2wasm", which is a part of the WebAssembly Binary Toolkit. Figure 1 show the process used to generate WebAssembly.

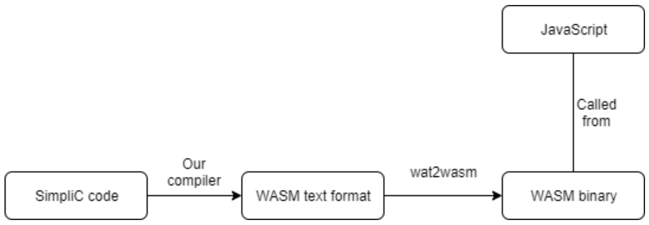


Figure 1. Compilation- and usage pipeline

The output of the compiler could be kept very short, had it not been the two I/O operations "print" and "read". When using the Emscripten compiler, including the *stdio.h*-header in C-programs would result in massive files, likely implementing these functions as WebAssembly functions.

An easy workaround is using the import system; It is possible to declare functions in WebAssembly to be external (similar to the keyword *extern* in C/C++), implementing them in JavaScript and importing them later on:

```
(import "imports" "print" (func $print (param i32)))
(import "imports" "read" (func $read (result i32)))
```

The above declares two functions, *print* and *read* to be external, and to be found in a Map-object under the key "imports".

In the JavaScript file, the following code creates the import object, including function pointers to print- and read respectively.

```javascript
function readfunction(){
    var value = prompt("Enter an integer:");
    return parseInt(value);
}

var importObject = {imports:
                        {print: arg => console.log
                            (arg},
                        read: readfunction}};
```

Listing 1. Importing JavaScript functions into WebAssembly

the object *importObject* may then be passed as a parameter when instantiating the WebAssembly module, see later sections.

[3]http://jastadd.org/web/

5 Evaluation

The evaluation of our work in WebAssembly may be divided into two parts; Benchmarking the execution speed of WebAssembly in comparison to JavaScript, and evaluating the integration of WebAssembly into JavaScript.

During both parts of the evaluation, the web browser Firefox was used.

5.1 Benchmarking

In order to benchmark WebAssembly, we needed to select some simple algorithm according to the single criterion of being expressible in SimpliC. This means our algorithm needs to be expressed only with integer arithmetic and stack allocation.

In the end, recursive fibonacci calculations without memoization was the algorithm of choice; the function's repeated call to itself makes the JIT-compiler activate after a certain threshold has been reached, optimizing the JavaScript code.

In order to get a better overview of the performance of WebAssembly, the following tests were made:

1. Average execution time over 10 iterations, JavaScript's JIT-compiler activated
2. Average execution time over 10 iterations, JavaScript's JIT-compiler deactivated
3. Average execution time over 10 iterations, using Emscripten to compile WebAssembly rather than our own compiler

For each of the above tests, the fibonacci numbers between 10 and 41 were calculated repeatedly, logging the average execution time of 10 iterations.

Furthermore, a different test was created specifically to measure how much the execution time varies in WebAssembly. The same fibonacci algorithm was used again. This time, in 30 iterations per language, a call to *fibonacci(35)* was done. After each iteration, the execution time was recored.

5.1.1 Test 1: JIT-active

As seen in figure 2, our compiler performed worst of all contestants. As one would expect Rust and Java are the two fastest languages when it comes to calculating fibonacci numbers. In fact they were only invited to the competition as two examples of performance-oriented languages. So what could be the cause leading to our wasm-code being slower than Javascript? We believe it mainly has to do with optimizations, which our compiler does not know how to perform. The fact that our compiler, without optimizations, still manages to be a worthy adversary to Javascript in this benchmark, is a good indication of how well WebAssembly performs.

The results of test 1 may be found in figure 2.

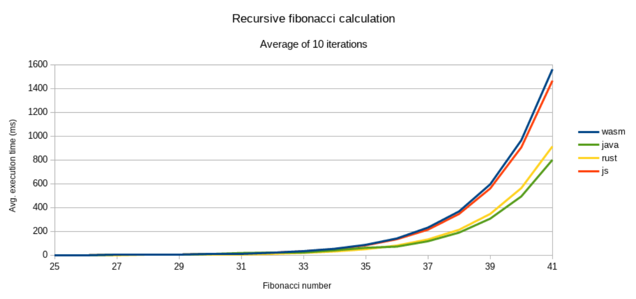


Figure 2. Performance of WebAssembly in comparison to various languages, Compiled by our SimpliC compiler

5.1.2 Test 2: JIT-inactive

When disabling the JIT-compiler in javascript our compiler outshines all of its opponents, i.e. Python and Javascript. Figure 3 shows the results of this test.

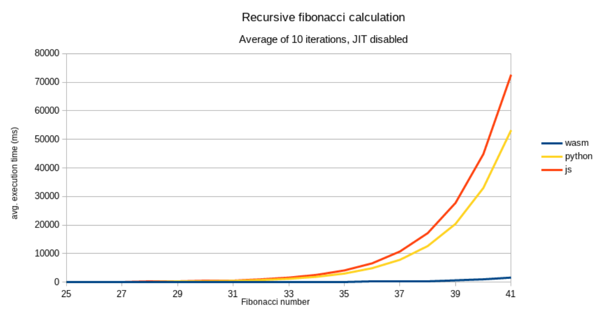


Figure 3. Performance of WebAssembly in comparison to various languages, Compiled by our SimpliC compiler, JIT-compilation disabled in JavaScript

5.1.3 Test 3: Emscripten compiled WebAssembly

Once the proper Emscripten compiler is allowed to compile our SimpliC fibonacci code the results become even more promising. In figure 4 it is clear that wasm is slightly faster. Even though we suspected a more dramatic decrease in time, compared to Javascript, it is nonetheless faster.

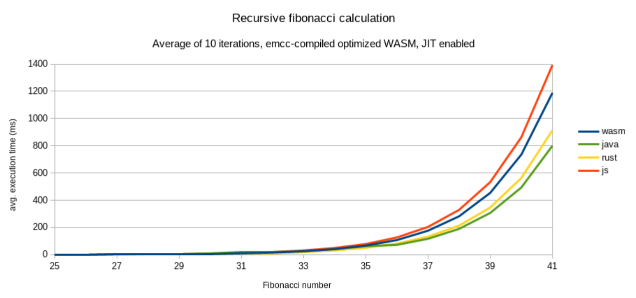


Figure 4. Performance of WebAssembly in comparison to various languages, Emscripten-compiled

5.1.4 Test 4: Variance

Calculating variance for a set of n samples is done with the following calculation:

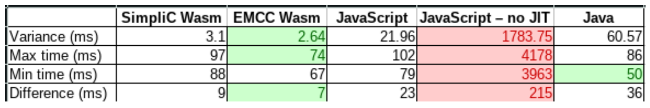
$$V(X) = \sum_{i=0}^{n} \frac{X_i - \overline{X}}{n-1}$$

Where $\overline{X}$ is the mean value of the set.

Figure 5 shows the result of this test, also highlighting the minimum- and maximum execution time for each language tested.

Variance can be thought of as a measurement of the distance between a randomly selected sample in the set and the mean value.

A large variance in this case means that the execution time varies more, while a small variance means the execution time tends to be similar between different runs.



	SimpliC Wasm	EMCC Wasm	JavaScript	JavaScript – no JIT	Java
Variance (ms)	3.1	2.64	21.96	1783.75	60.57
Max time (ms)	97	74	102	4178	86
Min time (ms)	88	67	79	3963	50
Difference (ms)	9	7	23	215	36

Figure 5. Variance, minimum- and maximum execution time of several languages

As can be seen, the variance for both SimpliC- and EMCC compiled WASM are the lowest of the tested languages. This is quite interesting, in the light of the previous tests where the execution speed was not as fast as initially believed. This effectively means that WebAssembly performs in a very predictable manner, compared to the second lowest variance in JavaScript, which is roughly tenfold of the WebAssembly variance.

5.2 Integration

Besides benchmarking, another important part of the evaluation was exploring both the ease-of-use of WebAssembly and its tools, and how to integrate it into JavaScript applications.

5.2.1 Generating WebAssembly

Using Emscripten would produce three quite large source files. calling the command *emcc main.c* produces:

a.out.wasm
a.out.js
a.out.html

The large size of the .wasm-file can be attributed to the code generated for handling allocation- and deallocation to the heap. Importing standard I/O functionality into C through the stdio.h-header would also generate code in order to support formatted prints through the C-function "printf".

The .html-file and .js-file which were also generated could be considered optional, since they make up a generated default webpage with a console, which runs your .wasm-binary as if it was run from a terminal window.

Besides the large files which were hard to navigate, we ran into two issues when generating WebAssembly in this manner:

1. **Function names are obfuscated.** Given a C-file with the sole function named "fibonacci", the produced WebAssembly file contains no such function, but rather a dozen functions following the naming convention "$f1, $f2... ".
2. **Functions are not exported by default.** This is not an issue if you only want to run the generated default website as it knows which function to call in order to start execution. However, combined with the previous point, whenever a custom website is to call some WebAssembly function, it becomes unfeasible to search for the desired function and export it manually.

The goal was to produce a very simple website which imported- and used a very simple WebAssembly binary, and the previously mentioned issues were a source of great frustration.

We did, however, discover a way to produce an accessible .wasm-binary without the generated website.

Given the same example as previously used, a C-file containing the sole function "fibonacci", the following command produces only a .wasm-binary, which also exports the function "fibonacci" without any name changing:

emcc main.c -o main.wasm -s EXPORTED_FUNCTIONS="['_fibonacci']"

5.2.2 Fetching WebAssembly from JavaScript

Given a WebAssembly binary "simple.wasm" and a function "exported_func" written in the source language, Mozilla's documentation for WebAssembly suggests the following way of calling exported WebAssembly functions:

```
var importObject = { imports: { imported_func: arg
    => console.log(arg) } };

WebAssembly.instantiateStreaming(fetch('simple.
    wasm'), importObject)
.then(obj => obj.instance.exports.exported_func())
    ;
```

Listing 2. Mozilla recommendation of calling WebAssembly

Here, we can see that the user passes the Map object "importObject" into the instantiateStreaming-function. This object contains all of the data and functions which are listed as imported inside of the WebAssembly binary. As previously mentioned in section 4, this passing method was used in order to use JavaScript functions to emulate both the "read" and "print"-functions in our generated WebAssembly code.

This way of executing WebAssembly functions was far from ideal for our purposes; it executes asynchronously, and requires some understanding of the intricacies of JavaScript in order to be fully utilized.

After a bit of tinkering, the following was found to work:

```
let exported_func;

  function loadWasm(filename){
    return fetch(filename, {mode: 'no-cors'})
    .then(response => response.arrayBuffer())
    .then(bits => WebAssembly.compile(bits))
    .then(module => { return new WebAssembly.
        Instance(module); });
  };

  loadWasm('simple.wasm')
  .then(instance => {
    exported_func = instance.exports.
        exported_func;
  });
```

Listing 3. Setting a function pointer to a WebAssembly function

By doing this, a function pointer to the WebAssembly function is saved in the JavaScript variable "exported_func", and may be called freely from that point onwards.

5.2.3 CORS request blocked

By loading a local resource with a call to the JavaScript function "fetch", as seen in the previous section, one may run into the error "CORS request blocked". CORS (Cross-Origin Resorce Sharing) is a security system used to prevent resources to be loaded on a website with a different origin (domain, protocol or port) than its own.

This problem can be solved in two ways:

1. Reaching the desired resource through an URL-path rather than the typical file path
2. Turning of CORS-request blocking.

We decided upon using the latter solution, which requires some tinkering with the web browser configuration.

6 Related work

The paper "*Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code*" has a similar research goal to our report. Through a similar test approach, it describes a process of using smaller, scientific programs of roughly 100 lines of code being compared in both WebAssembly and native code. In order to run UNIX applications in the web, they created the BROWSIX-WASM extension, which allows UNIX system calls to be done through the browser.

Their results are similar to ours; WebAssembly runs on average between 45 to 55% slower than native code depending on the browser, claiming that it is due to missing optimizations, and slowdown inherent to the WebAssembly platform [Jan+19].

In the paper "*New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild*", authors M. Musch et. al. investigated the spread of WebAssembly in 2019, and its' primary uses on the world-wide web.

It was discovered that in the Alexa Top 1 million websites, roughly 1500 of them execute at least some WebAssembly code. Around 50 % of these sites use WebAssembly for cryptomining and various malicious activity. The remaining sites use WebAssembly mainly in either cross-compiled C# in gaming applications, or as part of publicly available JavaScript libraries, intended to speed up execution.

It also suggests how to execute WebAssembly binaries through the JavaScript API, which were used as a clue for our understanding on how they integrate in our evaluation. [Mus+19].

Another paper which evaluates WebAssembly, "*Bringing the Web Up to Speed with WebAssembly*", gives an overview of the language, its' syntax and performance.

It reports that the performance, although varying heavily depending on the algorithm, has an upper bound execution time of 2x native code, and could even outperform it in a small set of cases. In this test suite, the size of the WebAssembly binaries are on average 85.3% of the native x86 binaries[Ros+nd].

7 Conclusion

As promising as it sounds on paper, WebAssembly has yet to overtake native code in execution time. It gives a slight boost to JavaScript whenever it is applicable, however both our results combined with the results of others show that the gap between native performance compared to WebAssembly is still noticable.

Although not easily quantifiable, our experience with learning to use- and using WebAssembly was not very smooth. Ideally, a user should be able to import- and use WebAssembly modules as if they were native JavaScript, letting the JavaScript engine take care of any of the internal mysteries of the language in the spirit of abstraction.

Today, however, a user is reliant on the asynchronous Promise-system of JavaScript, making the programmer required to not only have a working knowledge of one high-level language which compiles to WebAssembly, but also JavaScript.

WebAssembly is still a promising piece of technology. The variance test that was performed shows that it is quite predictable in its execution time in comparison to pure JavaScript. This can be benificial in applications that require predictable, smooth performance.

As of today it is only some two years old and obviously still in its infancy, but given some time we believe it could improve, potentially reaching speeds comparable to native code.

8 Future work

Several potential research topics emerged as this project progressed.

One interesting question that could be explored is whether there exists a significant difference in performance depending on which browser used. Microsoft Edge is, for instance, currently the only browser with Just In Time-compilation capabilities for WebAssembly. Could there also be a difference between Chrome and Firefox?

We were also worried that the use of only one algorithm as the base for all benchmarks would generate unreliable results. Using several algorithms for benchmarking would not only increase the reliability, but also perhaps find cases where Javascript outperforms WebAssembly.

For now our compiler outputs standalone WebAssembly code. The officially recognized compiler emcc does not only this but also generates the Javascript-glue as well as some html-files. In essence, the whole package needed for the script to run in the browser. And since the only purpose of WebAssembly is to be run in the browser it would prove useful to have our own compiler also generating the necessary Javascript and html.

References

[Ros+nd] Andreas Rossberg et al. “Bringing the Web Up to Speed with WebAssembly.” In: *COMMUNICATIONS OF THE ACM* 61.12 (n.d.), pp. 107–115. ISSN: 00010782. URL: https://doi-org.ludwig.lub.lu.se/10.1145/3282510.

[88] *RPC: Remote Procedure Call Protocol specification: Version 2.* RFC 1057. June 1988. DOI: 10.17487/RFC1057. URL: https://rfc-editor.org/rfc/rfc1057.txt.

[Tal+11] A. Taly et al. “Automated Analysis of Security-Critical JavaScript APIs.” In: *2011 IEEE Symposium on Security and Privacy, Security and Privacy (SP), 2011 IEEE Symposium on* (2011), pp. 363–378. ISSN: 978-1-4577-0147-4. URL: https://ieeexplore.ieee.org/document/5958040.

[Dai15] Hong-Yi Dai. “The Mysteries of Lisp - I: The Way to S-expression Lisp”. In: *CoRR* abs/1505.07375 (2015). arXiv: 1505.07375. URL: http://arxiv.org/abs/1505.07375.

[Jan+19] Abhinav Jangda et al. “Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 107–120. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jangda.

[Mus+19] Marius Musch et al. “New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment.* Ed. by Roberto Perdisci et al. Cham: Springer International Publishing, 2019, pp. 23–42. ISBN: 978-3-030-22038-9.

[Pro+19] Jonathan Protzenko et al. “Formally Verified Cryptographic Web Applications in WebAssembly.” In: *2019 IEEE Symposium on Security and Privacy (SP), Security and Privacy (SP), 2019 IEEE Symposium on* (2019), pp. 1256–1274. ISSN: 978-1-5386-6660-9. URL: https://ieeexplore-ieee-org.ludwig.lub.lu.se/document/8835291.