

Oron: Towards a Dynamic Analysis Instrumentation Platform for AssemblyScript

Munsters, Aäron Sarah-Louise; Scull Pupo, Angel Luis; Bauwens, Jim; Gonzalez Boix, Elisa

DOI:

[10.1145/3464432.3464780](https://doi.org/10.1145/3464432.3464780)

Publication date:

2021

Document Version:

Accepted author manuscript

[Link to publication](#)

Citation for published version (APA):

Munsters, A. S-L., Scull Pupo, A. L., Bauwens, J., & Gonzalez Boix, E. (2021). *Oron: Towards a Dynamic Analysis Instrumentation Platform for AssemblyScript*. 6-13. Paper presented at Fifth International Workshop on Programming Technology for the Future Web. <https://doi.org/10.1145/3464432.3464780>

Copyright

No part of this publication may be reproduced or transmitted in any form, without the prior written permission of the author(s) or other rights holders to whom publication rights have been transferred, unless permitted by a license attached to the publication (a Creative Commons license or other), or unless exceptions to copyright law apply.

Take down policy

If you believe that this document infringes your copyright or other rights, please contact openaccess@vub.be, with details of the nature of the infringement. We will investigate the claim and if justified, we will take the appropriate steps.

Oron: Towards a Dynamic Analysis Instrumentation Platform for AssemblyScript

Aäron Munsters, Angel Luis Scull Pupo, Jim Bauwens, and Elisa Gonzalez Boix
aaron.sarah-louise.munsters@vub.be, angel.luis.scull.pupo@vub.be, jim.bauwens@vub.be, egonzale@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

The dynamic nature of JavaScript may lead to challenges and issues regarding efficiency and security. Analysis tools can help developers tackle some of these issues. In the context of web applications, dynamic analyses are best suited for handling those dynamic features but may affect the programs execution performance. In a first experiment, we attempted to improve the performance of the Aran dynamic analysis platform for JavaScript by utilizing WebAssembly. The extension caused extra performance hits due to context switches between JavaScript and WebAssembly. Because these context switches are inevitable, we decided to refit our work for the analysis of AssemblyScript, a subset of TypeScript which compiles to WebAssembly (and therefore excluding context switches). In this work, we explore this approach in the form of a new source code instrumentation platform named Oron, which allows for the instrumentation of AssemblyScript code. The presented platform is evaluated and shows promising improvements which provide a solid basis for efficient dynamic analysis of AssemblyScript applications.

Keywords: web applications, dynamic analysis, source code instrumentation, WebAssembly, AssemblyScript

ACM Reference Format:

Aäron Munsters, Angel Luis Scull Pupo, Jim Bauwens, and Elisa Gonzalez Boix. 2018. Oron: Towards a Dynamic Analysis Instrumentation Platform for AssemblyScript. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/1122445.1122456>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ProWeb '21, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00
<https://doi.org/10.1145/1122445.1122456>

1 Introduction

JavaScript has become the language of client-side web applications by default. A key strength of JavaScript is in its dynamic nature, as it (i) does not require developers to code with type annotations, (ii) provides developers with a flexible prototype-based OOP framework, and (iii) allows developers to use runtime reflective capabilities to modify the applications execution state. While JavaScript's dynamic nature makes it suitable for fast prototyping and development, they complicate reasoning about an application's behaviour. Moreover, it may lead to challenges and issues regarding efficiency and security.

Analysis tools can help developers tackle these issues by providing insights and information on program code bases. A *static code analyser* analyses programs without actually executing them but rather by approximating their behaviour. Static analysis has shown to be useful for statically typed languages such as Java [10], C [18] and C++ [16]. They are, however, limited in their capabilities to reason about highly dynamic language features, such as those present in JavaScript [1]. An alternative to static analysis is *dynamic analysis*, in which the analysis is performed at run time. This is achieved by monitoring and analysing the execution of a program, instead of approximating its behaviours from code. Dynamic code analysers are therefore more suitable for languages with dynamic features than static analysers, as information on these features will be more readily available at run time. They are additionally useful for performing analysis on aspects that heavily dependant on the program's execution, e.g. such as performance analysis or concurrency analysis. The downsides of using dynamic analysis are that it may heavily affect the execution performance [13] and intended behaviour (i.e. the analysis is not transparent to the application) of the program. These effects may be problematic in certain types of analyses such as a security analysis, where malicious code can observe the application's altered behaviour and can consequently behave differently to avoid being discovered.

The main goal of this work is to explore an efficient source code instrumentation platform for web applications. Source code instrumentation is a suitable technique for implementing dynamic analysis in the web environment as it ensures the independence of the analysis from a specific JavaScript

111 engine (i.e., the analysis is portable across different JavaScript
 112 engines). This means that once a program is instrumented
 113 it can run fully self-contained. The main disadvantage of
 114 source code instrumentation is its performance overhead.

115 In this paper, we investigate the performance issues of Li-
 116 vail [5], a state-of-the-art dynamic analysis platform for web
 117 applications that makes use of source code instrumentation.
 118 To remediate some of them, we propose to use WebAssembly,
 119 an efficient low-level programming language that coexists
 120 with JavaScript in many platforms. In particular, we explore
 121 and implement a new source code instrumentation platform
 122 named Oron that allows instrumentation of AssemblyScript
 123 code, a subset of TypeScript (which is a superset of JavaScript)
 124 that compiles to WebAssembly. The presented platform is
 125 further evaluated and shows promising improvements which
 126 provide a good basis for efficient dynamic analysis on the
 127 Web.

130 2 Background and Motivation

131 Performing dynamic analysis on large scale applications
 132 often involves making use of general-purpose dynamic anal-
 133 ysis frameworks [4, 5, 12]. These frameworks remove the
 134 effort of developing analysis tools from scratch and allow
 135 developers to focus on the analysis itself.

136 Several different techniques are used by these platforms
 137 that can roughly be classified into four categories: bytecode
 138 instrumentation, AST instrumentation, meta-circular inter-
 139 preter, and source code instrumentation. While byte code
 140 and AST instrumentation allow for transparent analyses
 141 with a low-performance overhead, they are not portable as
 142 they require Virtual Machine modifications. Meta-circular
 143 and source code instrumentation, on the other hand, provide
 144 portable solutions that are best suited for web applications,
 145 but they incur a measurable performance overhead. In partic-
 146 ular, JavaScript source code instrumentation platforms have
 147 been reported to have a significant impact on application
 148 performance [5, 12].

149 Analysis using source code instrumentation work by *inter-*
 150 *cepting* program operations such as function calls, property
 151 writes, etc. For example, in Linvail [5], intercepting opera-
 152 tions can be done by inserting (weaving) special function
 153 calls (traps) around the program operations in the analysed
 154 application source code. Traps implement the analysis se-
 155 mantics and are defined in an *advice*.

158 2.1 Towards an efficient source code 159 instrumentation platforms

160 In order to improve the overall performance of an instru-
 161 mented application, one could try to optimize and reduce the
 162 analysis code. However, this may not always be feasible for
 163 JavaScript web applications, where it can be hard to reduce
 164 execution overhead. Another method for achieving better
 165

166 performance is through the means of WebAssembly [7]. We-
 167 bAssembly is a portable, safe and fast low-level binary code
 168 format that coexists with JavaScript in many web environ-
 169 ments¹. It offers high interoperability with JavaScript and
 170 serves as a compilation target for many other languages (e.g.,
 171 Rust, C, etc).

172 As a first step towards reducing execution overhead, we
 173 conducted some experiments starting from a JavaScript dy-
 174 namic analysis platform that relies on source code instru-
 175 mentation, Linvail [5]. In particular, we extended Linvail's
 176 source code instrumenter (called Aran) to use WebAssem-
 177 bly for analysis. Because compilation of JavaScript code to
 178 WebAssembly is not yet possible at this point in time, As-
 179 semblyScript was used instead as analysis language. Assem-
 180 blyScript [15] is a *strict* variant of TypeScript [2], which itself
 181 is a superset of JavaScript. As AssemblyScript targets and
 182 compiles to WebAssembly, it provides a suitable replacement
 183 for JavaScript considering the shared semantics and syntax.

184 The overall idea of the extension was to ensure that the
 185 application's source code could remained untouched (in
 186 JavaScript), while the analysis code would be in Assem-
 187 blyScript. Running Aran on an application would then re-
 188 sult in an instrumented application, along with the analysis
 189 compiled as a WebAssembly module which is linked to the
 190 instrumented application later at runtime. At runtime, any
 191 trapped operations would call into the advice (i.e., analysis)
 192 located in the WebAssembly module.

193 This strategy was evaluated by a series of benchmarks
 194 (based on a subset of the SunSpider benchmark suite²). How-
 195 ever, the results showed that using AssemblyScript to imple-
 196 ment the analysis for instrumented JavaScript applications
 197 performed worse (i.e., 2.37x to 107.98x slowdown) than using
 198 JavaScript to implement the analysis (as Aran does). After
 199 some research, it became clear that the problem was related
 200 to context switches between JavaScript and WebAssembly.
 201 The overhead of these context switches was greater than the
 202 benefit that could be obtained by utilising WebAssembly.

203 Because our extension to Aran did not allow for an imme-
 204 diate performance improvement on the analysis of JavaScript
 205 programs, we decided to refit our work for analysis on As-
 206 semblyScript programs and evaluate the performance of this
 207 approach instead. By using AssemblyScript as the target lan-
 208 guage, no context switches will occur. Additionally, because
 209 JavaScript and AssemblyScript are closely related, many
 210 JavaScript programs can be compiled to AssemblyScript with
 211 minor changes, allowing us to compare approaches. In the
 212 next sections, we describe, evaluate, and discuss this ap-
 213 proach in the form of a new dynamic analysis platform.

¹WebAssembly is a W3C standard already being supported by all major
 browsers, Node.js, and it can also run outside the web context.

²https://wiki.mozilla.org/Sunspider_Info

3 Oron: a Dynamic Analysis Platform for AssemblyScript

In this section, we present Oron, a dynamic analysis platform based on source code instrumentation for web applications written in AssemblyScript. Figure 1 shows the overall architecture of Oron, which is based on the architecture of Aran. In Oron, the target application and analysis are both written in AssemblyScript, enabling the compilation of the instrumented application to WebAssembly. In theory, the application itself can be written in any high-level language that can compile to WebAssembly, but as web applications are typically developed in JavaScript/TypeScript, AssemblyScript should not pose a fundamental barrier.

The rest of this section we first describe Oron and its interface by implementing a profiling analysis, using the Ackerman function as input program. Then we explain how to deploy such instrumented application.

3.1 Instrumenting function applications

Consider as example application, the Ackermann recursive function [3] shown in Listing 1. A call to the main function on line 11 will perform a call to the ackermann function with arguments 4 and 5, resulting in a set of recursive calls eventually returning the result of the calculation.

```

1 function ackermann(m: i32, n: i32): i32 {
2   if (m == 0) {
3     return n + 1;
4   } else if (n == 0) {
5     return ackermann(m - 1, 1);
6   } else {
7     return ackermann(m - 1, ackermann(m, n - 1));
8   }
9 }
10 export function main(): i32 { return ackermann(4, 5); }

```

Listing 1. An AssemblyScript program implementing the Ackermann function being called in its main function.

Writing an analysis for profiling the execution of this program is done by extending the `OronAnalysis` interface provided by Oron. Listing 2 shows a profiling analysis for the Ackerman function. As analysis developers, we are interested in intercepting calls to the `ackermann` function in order to maintain the counter. This is done by implementing the `preApply` trap from the `OronAnalysis` interface. Within the trap's body, analysis developers have access to the runtime information of the operation being intercepted.

For example, in the `preApply` implementation of Listing 2 the developer can access the function pointer, name and arguments of the function being called. Specifically, the profiler analysis (see lines 6 to 12) keeps track of the number of calls of the `ackermann` function and the number of times the arguments provided to the function are equal to zero.

Besides the `preApply` trap used in this analysis example, Oron provides a wide number of traps for intercepting the

program operations. A non-exhaustive list of the supported program operations and their corresponding trap is listed below:

- `preApply`: Function applications before the function is called
- `postApply`: Function applications after the function has been called
- `propertyAccess`: Instructions in which an objects property is being read
- `propertySet`: Instructions in which an objects property is being written

The implementation of the traps is selective, meaning that developers are not forced to implement all traps during the implementation. Oron will only use the implemented traps to perform the instrumentation. These traps can be combined in different ways to measure certain properties of the instrumented program, such as recursion depth, function profiling, object profiling and different combinations depending on the input program.

```

1 let count: i32 = 0, zeroes: i32 = 0;
2 export function getCount(): i32 {return count;}
3 export function getZeroes(): i32 {return zeroes;}
4
5 export class MyAnalysis extends OronAnalysis {
6   preApply(fname: string, fptr: usize, args: ArgsBuffer):
7     void {
8     if (fname === "ackermann") {
9       count++; // increment counter
10      if (args.getArgument<i32>(0) === 0) zeroes++;
11      if (args.getArgument<i32>(1) === 0) zeroes++;
12    }
13 }

```

Listing 2. Definition of an Oron analysis to profile the amount of function calls and determine the amount of zeroes.

3.2 Deploying an Analysis

After the analysis has been defined, it still needs to be linked (i.e., instrument the program) to the program and compiled to WebAssembly. For the instrumentation Oron requires both, the input program (e.g., Listing 1), the analysis program (e.g., Listing 2) and the output path to output the instrumented program, as illustrated below:

```
1 node oron.js ackermann.ts analysis.ts instrumentedAck.ts
```

This will output the file `instrumentedAck.ts` containing the instrumented code of `ackermann.ts` with the analysis applied defined in `analysis.ts` in a single functional program.

During the instrumentation, Oron will traverse the AST of the input program and will instrument all nodes for which a trap was defined in the analysis implementation. For example, in Listing 2 it means that AST nodes corresponding to function calls in the lines 5, 7, and 10 will be instrumented.

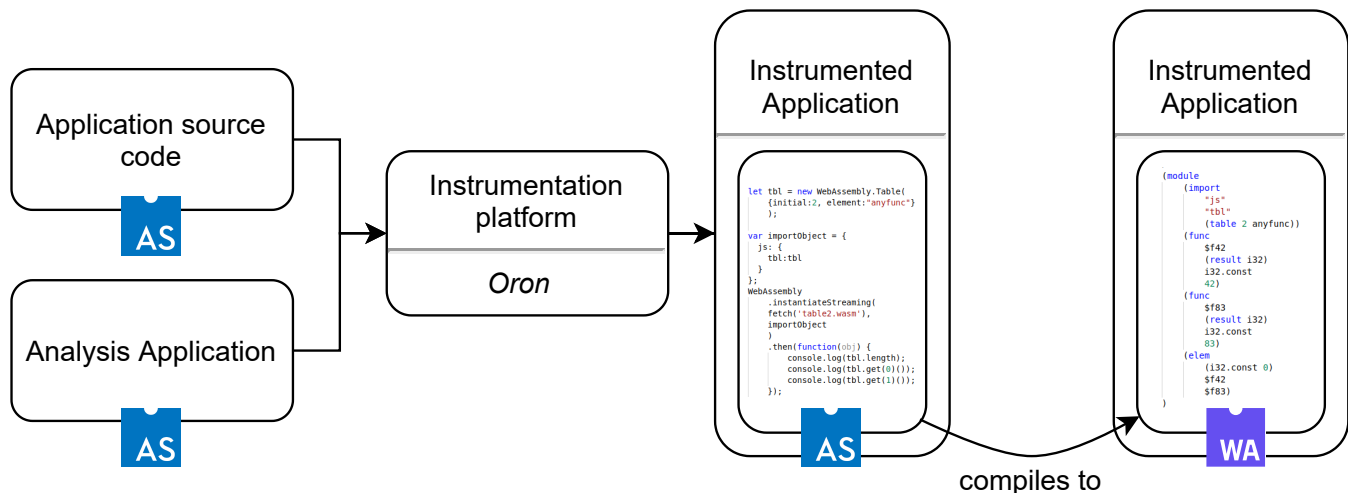


Figure 1. Overall architecture of the Oron AssemblyScript instrumentation platform.

The output instrumented program can be then compiled into a WebAssembly program by making use of the AssemblyScript compiler.

After compiling the instrumented program, the program can be instantiated in a JavaScript program as module. Once the module instance has been loaded, the main function can be invoked to run the original AssemblyScript application. In our running example, after the main execution, a call to `getCount` and `getZeroes` would return the amount of function calls and amount of zeroes in the program during the invocation of the main function.

3.3 Instrumenting the running example

Given the earlier code example enlisted in Listing 1 and the accompanying analysis enlisted in Listing 2, we now briefly describe the transformation that Oron perform to produce the instrumented code for an Ackermann function call.

As shown in Listing 2, the analysis method of performing a `preApply` is invoked with three arguments: the function name `fname`, a pointer `fptr` to the function and a data structure `args` containing the arguments and their type information. This is done to allow maximum flexibility to the analysis developer, as the function name allows to perform a string match for the searched function, the function pointer is to allow the analysis to store a reference to the function for later retrieval and the arguments are provided to inspect them before the function application will take place.

Listing 3 shows the transformation that Oron performs to account for the arguments to be passed to the function call `ackermann(4, 5)`.

```

5 MyAnalysis.preApply(
6   "ackermann",
7   changetype<usize>(ackerman),
8   args)
9 );
10 // call to function
11 ackerman(4, 5);

```

Listing 3. The transformation performed by Oron for a function call.

The `argbuffer` is a globally accessible buffer containing the values of the arguments passed to the instrumented function. More concretely, it is a raw binary buffer that reserves the maximum amount of memory needed for the arguments passed to any function call in the base program, which can be statically determined. Besides a copy of the arguments, the buffer stores information on the types for the arguments. This type information would otherwise be irretrievable for the analysis developer after compilation. Additional discussion regarding the resolution of type information is elaborated in Section 4.

Storing the arguments requires their types to be known as specified by the AssemblyScript `store` instruction. As such, the Oron implementation takes a look at the function signature of the `ackermann` function, in this case, being twice an argument of type `i32`. Using `setArgument`, the type information and memory for the argument is stored in the buffer as shown in lines 2-3.

4 Implementation

This section provides some implementation details on the Oron platform. As mentioned before Oron manipulates AST nodes of the source code program in AssemblyScript. The section provides implementation details on how to obtain

the AST nodes and transform them to produce the final instrumented output source code (which is then compiled to WebAssembly).

Obtaining the input program AST. Accessing the AssemblyScript input code AST could be done making use of the AssemblyScript compiler API. The AssemblyScript compiler provides means of hooking into the compilation process with a “--transform” flag during compilation. With this flag a script is provided that gains access to inspect and modify the AST before it is further processed in the compilation pipeline. Alternatively, one can make use of the TypeScript compiler API. Since AssemblyScript is a variant of TypeScript, AssemblyScript programs can be parsed by the TypeScript compiler. The TypeScript compiler allows traversing the AssemblyScript compiler as if it were a TypeScript program AST, though it is essentially the same from a point of code structure.

Both options were considered for Oron, and the current prototype implementation employs the TypeScript compiler as it facilitates the AST traversal (as detailed below) and it supported type inferencing (required for instrumenting function calls).

AST Traversal. Once having the program AST representation as a TypeScript AST, the TypeScript compiler API provides operations to traverse the AST making use of the visitor pattern. The API provides a clear interface facilitating how AST node matching with a syntactical structure takes place. These features are currently not present within the AssemblyScript compiler API, making the TypeScript compiler API a more viable way to traverse the program AST.

Transforming AST nodes. During the instrumentation, certain target nodes should be transformed, which means that new instrumented nodes should be created and replace those target nodes. Doing so making use of the AssemblyScript API requires implementing the expected AST node generators and adjusting the traversing method to allow replacing the target nodes within the provided AST. Unfortunately, such traversal features combined with the generation features are not present within the AssemblyScript compiler API, which entails manual implementation and maintenance during further development of Oron.

In contrast to the AssemblyScript API, the TypeScript compiler API provides a clear interface with AST node factories for each type of node. Moreover, the traversal method provided by this API includes means to effectively replace the target nodes. Therefore, AST nodes transformation within the Oron is fulfilled using the TypeScript compiler API.

Resolving type information. As discussed in Section 3.3, certain steps need to be followed before retrieving information from AST nodes. AST nodes themselves do not contain type information. Instead, this information needs to

be resolved by TypeScript’s type-checker. AssemblyScript includes several types that are not natively included in TypeScript (such as `i32` and `u64`). The AssemblyScript compiler will compile these types directly to the underlying WebAssembly types. The TypeScript Compiler API has to be made aware of these types before any instrumentation can be performed, otherwise, it will assume that it is traversing the AST of a JavaScript application. If this is the case, any AssemblyScript specific type will resolve to TypeScript’s “any”-type and important type information will be lost.

Beside type information, it is also important to inform the TypeScript compiler about native AssemblyScript functions and their signatures. Without this information, the TypeScript compiler would resolve all functions to any. The AssemblyScript repository contains an up-to-date file with all primitive AssemblyScript definitions. A version of this file is used by Oron to guide the TypeScript compiler during AST transformations. It contains modifications that ensure that the native types of AssemblyScript are not resolved to valid (conflicting) JavaScript types, retaining their information after transformations by Oron.

Producing the output transformed source code. Finally, Oron outputs the transformed AST code into an AssemblyScript file which is then compiled to WebAssembly. Besides using the TypeScript compiler API for generating the AST, traversing and transforming it, it is also used to output the source code. The API also provides a printer, instructed to output the AST transformation to the output file specified by the Oron caller, effectively rounding up the instrumentation process.

5 Performance Experiments

In this section, we do a first validation of Oron as a source code instrumentation platform by running several performance benchmarks that aim to answer to the following questions:

RQ1: What is the runtime overhead introduced by the instrumentation?

RQ2: What factor of code size increase can be expected due to the instrumentation?

As discussed in Section 2, performance overhead is considered to be the main disadvantage of source code instrumentation. Besides overhead, it is also necessary to quantify the expected increase in source code size. WebAssembly is mainly used in an effort to improve overall performance. However, differences between the original source code and the final compiled binaries may be large enough to matter in terms of network load and resource usage. We present the results for both research questions in Section 5.2 and Section 5.3, respectively.

5.1 Methodology

To provide an answer on these research questions, we ran experiments on a Dell XPS with an Intel Core i7-8550U CPU @ 1.80Ghz with 16GB of RAM. For all benchmarks we deploy Oron on Node.js 14.2.0 and each individual benchmark was run 30 times. For each of these runs, the average execution time is used to model performance. These averages are then used to compute the slowdown between the original program and its instrumented version.

The evaluation uses three profiling analyses, each instrumenting different program operations, and an empty analysis. More concretely, the profiling analyses employ `functionCalls`, `propReads` and `propWrites`. These analyses count (i) the number of function calls to a certain function, (ii) the number of object property accesses for objects and their properties and (iii) the amount of object property writes, respectively. The empty analysis is included to measure the overhead introduced by the analyses over the base program.

The analyses were tested on a subset of the SunSpider benchmark suite³. We translated an existing implementation of the benchmarks for TypeScript⁴ to AssemblyScript.

5.2 Evaluating the runtime overhead

Figure 2 plots the results of each benchmark, visualizing the performance overhead and slowdown introduced by Oron. In the following we discuss some preliminary conclusions drawn from these benchmarks results.

Worst slowdown factor. The input program `bitops-3bit-bitsin-byte` shows the largest slowdown, of 104x, for profiling function calls. The program depends on a relatively large amount of function calls to a single particular function that performs a set of byte code operations (left bit shifts, right bit shifts). As the analysis is instructed to profile each function call, the relative slowdown of a CPU optimized operation (bit shift) compared to the high-level operations introduced by the analysis (string comparison, data structure manipulation) is to be expected.

In short, an increased slowdown can be expected when the input program only performs low-level operations. As Oron's injected instrumentation code uses high-level constructs, the injected code would outweigh the original program and introduce a large overhead.

Slowdown factor for function calls. In this section we take a look at benchmarks that involve a high number of function calls. In particular, `controlflow-recursive` shows a slowdown of 72x. The input program has many invocations to the ackermann function [3], the fibonacci function [6]

³https://wiki.mozilla.org/Sunspider_Info

⁴<https://github.com/apurvaraman/sunspider-jsx/tree/master/js/tests/sunspider-1.0/ts>

and the `tak` function [11]. These functions involve a high number of recursive function calls, and are used to benchmark the recursion capabilities of AssemblyScript. As was shown in Section 3.3, every instrumented function call is accompanied with operations that store its arguments in a data structure (provided by Oron) A possible cause of the large slowdown factor might be related to the additional call stack growth and the required argument buffer fill instructions as this is repeated many times in the recursive function calls.

Slowdown for property reads and writes. The slowdown introduced by `propReads` and `propWrites` is fairly small throughout most benchmarks, except for `access-nbody`. Inspecting the programs, we can see that `access-nbody` is the only program in the benchmark suite making use of AssemblyScript classes and objects. As such, it is clear the overhead is not present on benchmarks that do not use classes and objects. This overhead is mainly due to the extra level of indirection added by the instrumentation during property read and write operations.

Slowdown on instrumentation with empty analysis.

In this section we look at the results of the benchmarks with the empty (`identity`) analysis. There is one aspect that was observed during the benchmarks that is worth discussing, however, additional experimental data will be needed to any draw conclusion. An excessive performance cost for instrumenting function calls can be observed during the `identity` analysis on the programs `bitops-3bit-bits-in-byte`, `bitops-bits-in-byte`, `controlflow-recursive` and `math-spectral-norm`, while this same analysis had little to no impact in `access-nbody`, which has a very low number of function calls when compared to other operations. This excessive impact of just trapping an operation may be the result of manipulating the global arguments buffer, which we plan to solve in the future by statically analysing the traps and ensuring that the transformation only selectively prepares the arguments if they are used in the trap.

Overall conclusions. From the benchmarks conducted, we conclude that the additional overhead in AssemblyScript applications ranges from 1X to 104X slowdown. However, comparing the performance overhead of the `identity` analysis between Oron and the combination of Aran and WebAssembly (done in Section 2), we found that our approach performs better in general, ranging from 1x to 25x overhead in Oron versus 2,37x to 107.98x of the combination of Aran and WebAssembly.

5.3 Evaluating the code size increase

We now evaluate our approach with respect to the increase in code size for the instrumented programs. At a first glance, code size could be estimated by means of the lines of code or characters of code. This, however, makes less sense in the context WebAssembly which has a binary format. We

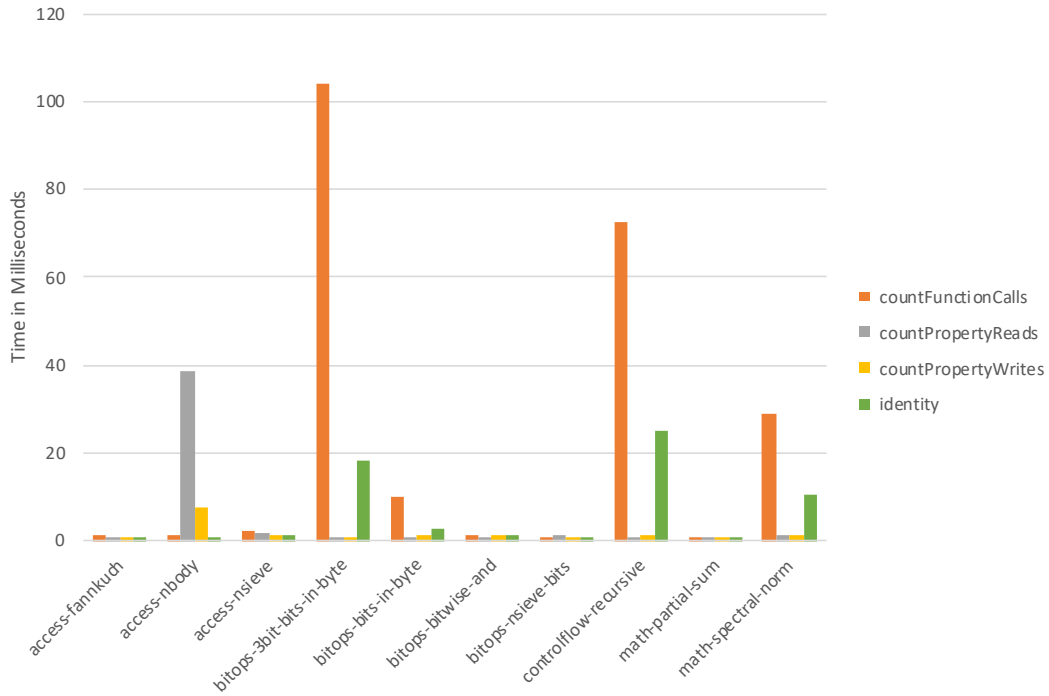


Figure 2. Slowdown factor on a partial set of the SunSpider benchmarks for four analyses built on top of Oron

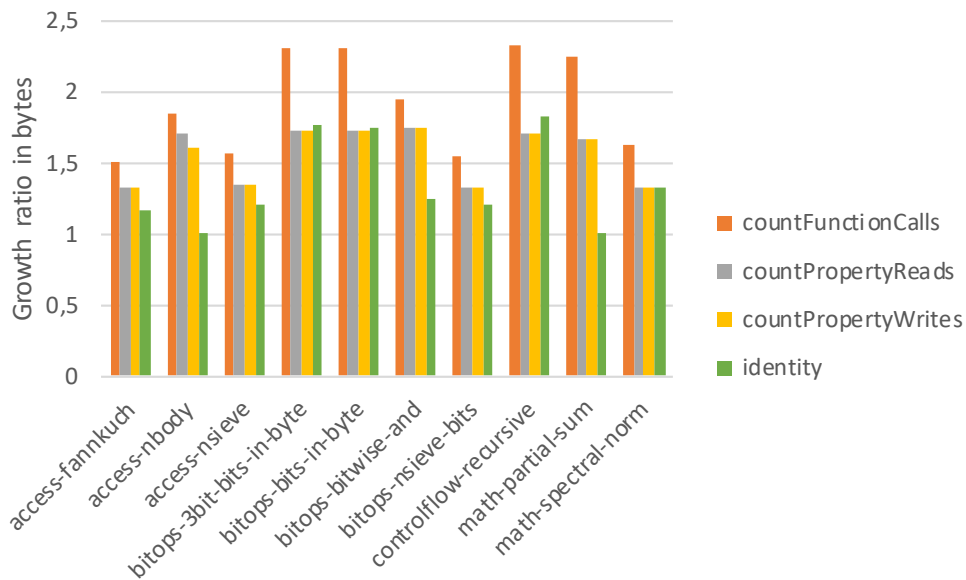


Figure 3. Instrumented source code growth ratio measured in bytes.

therefore focus on the increment of byte code as is presented in Figure 3.

As can be seen in Figure 3, binary sizes of instrumented applications of our benchmarks program have a grow ratio between 1x and 2,3x in the number bytes compared to the

original base program. This growth depends on the traps implemented in the analysis and the frequency of such trapped operations in the analysed program.

6 Related work

This section describes the closest related work to Oron in the context of web applications.

Wasabi [9] is a general-purpose dynamic analysis framework for WebAssembly. The framework works by inserting calls to JavaScript functions in the binary representation of WebAssembly. In contrast to Wasabi, where the analysis targets WebAssembly code (and therefore low-level program operations), in our approach the instrumentation targets AssemblyScript code (i.e., a set of higher-level program instructions).

NodeProf [14] is an instrumentation platform for JavaScript application running on Graal.js [17]. A distinguishing feature of NodeProf is that applications are instrumented at the AST-level, i.e. nodes of the AST can be wrapped with additional operations that perform the analysis. This allows the analysis to be enabled or disabled at runtime, with no overhead when disabled. However, as most JavaScript engines do not allow access to the AST, NodeProf cannot be used outside the Graal.js ecosystem.

Another known technique would be to make use of a meta-circular interpreter which implements the JavaScript interpreter in JavaScript itself. This allows for full control and visibility of the program as the interpreter has been redefined, as shown in Photon [8]. This technique has the advantages of being portable across platforms as it does not rely on a new interpreter and allows for full transparency to the original application.

These advantages, however, come at the cost of slowdown execution due to the overhead of the additional engine and loss of just-in-time compilation. Another disadvantage is that upon changes in the specification of the language, the meta-circular interpreter consequently requires updates to support these changes.

7 Conclusion

This paper presented Oron, an instrumentation platform designed for AssemblyScript, a popular high-level language in the JavaScript ecosystem used to compile to WebAssembly. The Oron platform instruments AssemblyScript programs by inserting traps to perform an analysis developed in AssemblyScript. It performs this instrumentation by using the TypeScript compiler API which can handle AssemblyScript programs as they are valid TypeScript. The resulting instrumented application is then compiled to WebAssembly and can be executed in any compliant engine.

Oron solves challenges regarding argument passing which are unknown at compile time by dynamically manipulating a runtime buffer with a set of arguments allowing the analysis to inspect runtime values. Evaluation of Oron with compute-intensive benchmarks programs shows a 1X to 104X runtime overhead. These results show that Oron exhibits less performance overhead in our benchmarks than a combination of

Aran and WebAssembly. However, our experiments comparing these approaches are still too limited to give a general conclusion.

Our future work is twofold: (i) we would like to conduct benchmarks to gain more insights in the expensive operations added by Oron and, (ii) explore a different strategy to reduce overhead of instrumentation of function calls operations by looking into a more efficient strategy for handling arguments.

References

- [1] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–36.
- [2] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on Object-Oriented Programming*. Springer, 257–281.
- [3] Cristian Calude, Solomon Marcus, and Ionel Tevy. 1979. The first example of a recursive function which is not primitive recursive. *Historia Mathematica* 6, 4 (1979), 380–384.
- [4] Laurent Christophe. 2015. Aran. <https://github.com/lachrist/aran>. Last captured on 5th, January 2020.
- [5] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. 2016. Linvail: A general-purpose platform for shadow execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 260–270.
- [6] Edsger W Dijkstra. 1979. In honour of Fibonacci. In *Program Construction*. Springer, 49–50.
- [7] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 185–200.
- [8] Erick Lavoie, Bruno Dufour, and Marc Feeley. 2014. Portable and efficient run-time monitoring of javascript applications using virtual machine layering. In *European Conference on Object-Oriented Programming*. Springer, 541–566.
- [9] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Providence, RI, USA) (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1045–1058. <https://doi.org/10.1145/3297858.3304068>
- [10] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *USENIX Security Symposium*, Vol. 14. 18–18.
- [11] John McCarthy. 1979. An interesting lisp function. *ACM Lisp Bulletin* 3 (1979), 6–8.
- [12] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.
- [13] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision.. In *USENIX Annual Technical Conference, General Track*. 17–30.
- [14] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction*. ACM, 196–206.
- [15] The AssemblyScript Project. 2020. AssemblyScript. <https://www.assemblyscript.org/>. Last captured on 9th, August 2020.

881	[16] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. 2000.	and Matthias Grimmer. 2017. Practical Partial Evaluation for High-	936
882	ITS4: A static vulnerability scanner for C and C++ code. In <i>Proceedings</i>	Performance Dynamic Language Runtimes. <i>SIGPLAN Not.</i> 52, 6 (June	937
883	<i>16th Annual Computer Security Applications Conference (ACSAC'00)</i> .	2017), 662–676. https://doi.org/10.1145/3140587.3062381	938
884	IEEE, 257–267.	[18] Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. A memory	939
885	[17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas	model for static analysis of C programs. In <i>International Symposium On</i>	940
886	Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon,	<i>Leveraging Applications of Formal Methods, Verification and Validation</i> .	941
887		Springer, 535–548.	942
888			943
889			944
890			945
891			946
892			947
893			948
894			949
895			950
896			951
897			952
898			953
899			954
900			955
901			956
902			957
903			958
904			959
905			960
906			961
907			962
908			963
909			964
910			965
911			966
912			967
913			968
914			969
915			970
916			971
917			972
918			973
919			974
920			975
921			976
922			977
923			978
924			979
925			980
926			981
927			982
928			983
929			984
930			985
931			986
932			987
933			988
934			989
935			990