Aran: JavaScript Instrumentation for Heavyweight Dynamic Analysis

Laurent Christophe[1], Coen De Roover[1] and Wolfgang De Meuter[1]

[1]*Software Languages Lab, Vrije Universiteit Brussel, Belgium*

Abstract

We introduce ARAN, a novel approach for developing heavyweight dynamic analyses of JavaScript programs. Our approach is broadly applicable, relying solely on source code instrumentation. It also supports complex customization through an intuitive aspect-oriented API, which consists of 31 entry points. With about 300 lines of code, our API can express an analysis that computes the symbolic execution tree of run-time values. Importantly, our approach preserves the semantics of the program under analysis. We implemented this approach in a tool and validated it using TEST262, the official JavaScript conformance test suite, achieving a 99.7% success rate and observing performance slowdowns rarely exceeding 10x.

Keywords

JavaScript, Dynamic Program Analysis, Shadow Execution, Source Code Instrumentation, Aspect-Oriented Programming

1. Introduction

JavaScript is a popular and versatile programming language that is the de facto standard for web development. It is also infamously known to be hard to analyze statically due to its dynamic nature. And so, insights into the behavior of JavaScript programs must often be gathered dynamically. In practice, only so-called "lightweight" dynamic program analyses, such as code coverage and profiling, are commonly deployed. Nonetheless, there exists another class of so-called "heavyweight" dynamic analyses. Examples include: taint analysis which is capable of detecting violations of security policies, and concolic testing which is capable of automatically generating test cases. These analyses share the need to track values at run-time especially primitive values – e.g., tracking strings such as password data is crucial for taint analysis. This is commonly achieved through a technique called *shadow execution*, which involves mirroring part of the program state with meta information. Although these heavyweight dynamic analyses have their merits, they are rarely used in practice. We hypothesize three main reasons for this lack of adoption:

Narrow Applicability: Academic approaches are often unsuitable for real-world deployment because they depend on a modified JavaScript runtime. While this may be effective for controlled experiments on a specific corpus of programs, it is typically impractical for real-world scenarios. JavaScript is deployed across a diverse and rapidly evolving ecosystem of runtimes, both on the client and server sides. For broader adoption, analyses must be **applicable** across a wide range of use cases.

Complex Customization: Heavyweight dynamic analyses are inherently complex and demand a high level of customization to yield meaningful results. For example, taint analysis involves specifying sources and sinks, while concolic testing requires mechanisms to generate new inputs. To encourage adoption, such analyses must offer an expressive interface that supports extensive customization while remaining concise and easy to understand.

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium
laurent.christophe@vub.be (L. Christophe); coen.de.roover@vub.be (C. De Roover); wolfgang.de.meuter@vub.be (W. De Meuter)
https://lachrist.github.io (L. Christophe); https://soft.vub.ac.be/~cderoove (C. De Roover)
0009-0001-5294-0585 (L. Christophe); 0000-0002-1710-1268 (C. De Roover); 0000-0002-5229-5627 (W. De Meuter)

Compromised Correctness: It is crucial for developers to trust the results of the analysis. Commonly, errors arise when the meta layer of the analysis interferes with the base layer of the program under analysis. In other words, the analysis must remain invisible to the program and preserve its original semantics – a property we refer to as *transparency*.

In this paper, we present ARAN, a generic approach for building heavyweight dynamic analyses. Our approach is broadly applicable by design, as it avoids relying on a modified JavaScript engine and instead uses source code instrumentation. Additionally, it addresses the inherent complexity of heavyweight dynamic analyses by offering an aspect-oriented API. We argue that the necessary level of customization for such analyses can only be achieved through programmable logic, rather than relying on simple configuration data.

The primary contribution of this paper lies in its demonstration that aspect-oriented programming, implemented through source code transformation, is well-suited for building heavyweight dynamic analyses of JavaScript programs. First, it offers expressiveness: our API enables shadow execution of complex JavaScript code with 31 entry points which is manageable. Notably, it supports expressing an analysis that computes the symbolic execution tree of run-time values in just 300 lines of code. Second, it achieves transparency: we implemented our approach in a tool and validated it against TEST262, the official JavaScript conformance test suite, achieving a 99.7% success rate with performance slowdowns rarely exceeding 10x.

The rest of this paper is organized as follows. In Section 2 we provide an overview of the approach and discuss its applicability. In Section 3, we present an intermediate language for protecting the API of our approach from the complexity of JavaScript. In Section 4, we present an aspect-oriented API suitable for building shadow execution and discuss the expressiveness of our approach. In Section 5, we evaluate the transparency of our approach by validating it against TEST262.

2. ARAN's Overview

The main idea behind ARAN is to express analysis as logic. So during analysis, two layers will coexist inside the same JavaScript program: the layer of the program under analysis, which we call the *base* layer and the layer of the analysis, which we call the *meta* layer. A natural way to weave the logic of the analysis into the target program is to rely on *aspect-oriented programming* [1] which entails expressing the logic of the analysis in JavaScript as well. In this paradigm, behaviors which are collectively called *advice* are inserted at execution points called *join points* according to a specification called *pointcut*. This action is called *weaving*. The combination of an advice and a pointcut is called an *aspect*. Hence, in our approach, an analysis is expressed as an aspect. In Listing 1, we present a simple aspect that logs the call stack of a program by inserting logic around every function application of the target program.

ARAN focuses on weaving the user analysis into the target program by performing source code transformation while the actual deployment is left to the user. This is a design choice that we made to ensure that the approach remains broadly applicable. Indeed, integrating an analysis into a build system or a CI/CD pipeline is a non-trivial task and goes beyond the scope of this paper. We envision two main architectures for deploying ARAN.

First is *offline* deployment, which is depicted in Figure 1. In this architecture, the analysis happens in two separate processes that run sequentially. In the first process, the target program is instrumented based on the pointcut of the analysis, and the setup code is generated. Then, these two parts are bundled with the advice of the analysis, which creates a standalone JavaScript program. The analysis will be performed once this program is executed as it would have been originally. This architecture is useful to reduce the memory footprint and performance overhead of the analysis.

Second is *online* deployment, which is depicted in Figure 2. In this architecture, everything happens in a single JavaScript process. That is, the runtime must be parameterized to preload ARAN and instrument the target program at load-time. This architecture is more likely to impact the behavior of the target program and requires engine parameterization, but it enables instrumentation of dynamically generated code. While global code does not necessarily need to be instrumented, code evaluated by a direct `eval`

```javascript
export const initialState = { depth: 0 };
export const aspect = {
  pointcut: ["apply@around"],
  advice:  {
    "apply@around": (state, callee, thisArgument, argumentList, path) => {
      state.depth += 1;
      try {
        console.log(".".repeat(state.depth) + " >> " + callee.name + " at " + path);
        const result = Reflect.apply(callee, thisArgument, argumentList);
        console.log(".".repeat(state.depth) + " << " + callee.name + " at " + path);
        return result;
      } catch (error) {
        console.log(".".repeat(state.depth) + " !! " + callee.name + " at " + path);
        throw error;
      } finally {
        state.depth -= 1;
      }
    },
  },
};
```

Listing 1: A simple analysis that records the call stack of programs.

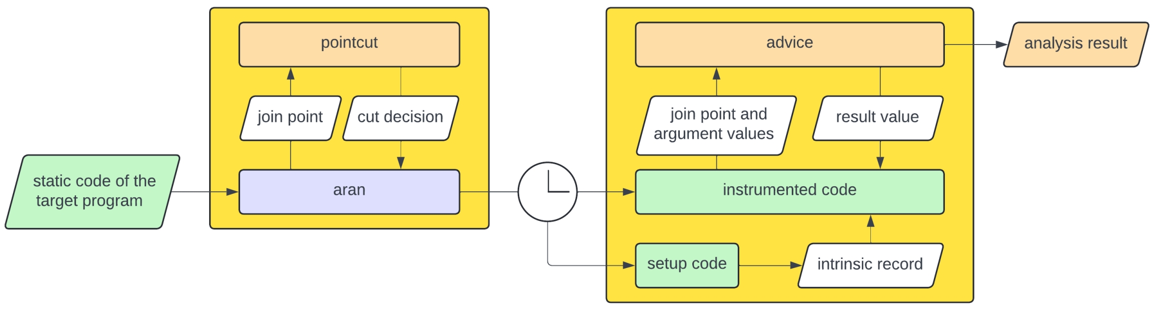

Figure 1: Offline architecture for deploying ARAN.

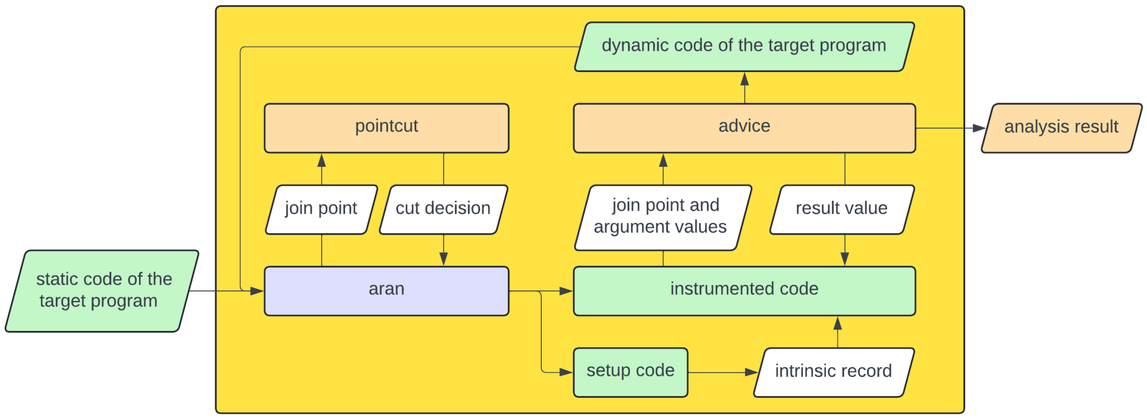

Figure 2: Online architecture for deploying ARAN.

call imperatively does. Otherwise, the base layer of the target program will be able to access the meta layer of the analysis. It follows that direct `eval` calls are only supported in this architecture.

We are aware that these architectures are work-intensive for the user. In the future, we plan to investigate how relevant functionalities could be provided for common workflows. For instance, offline architecture could benefit from a CLI that would expect a selection of files to instrument and would bundle the standalone JavaScript program. And the online architecture could benefit from a tool able to parameterize various JavaScript runtimes to instrument the target program at load-time.

3. ARAN's Intermediate Language

Instead of directly weaving the analysis logic in JavaScript, our approach performs weaving in an intermediate language instead. If weaving were to be performed directly in JavaScript, the aspect-oriented API would require such a high number of different join points that it would make the approach hard to customize. This is due to the fact that JavaScript has become quite a complex language over the years. Indeed, since 2015, the TC39 committee, which is in charge of maintaining ECMAScript – i.e., the de facto standard specification of JavaScript – has accelerated the rate at which new features are introduced and switched to a yearly release cycle. While this has kept JavaScript fresh and relevant, it has also rendered it more difficult to analyze. Our intermediate language is called ARAN-LANG and was designed with the following three criteria in mind:

Minimal: ARAN-LANG should be as simple as possible to enable the user to express complex analyses that require shadow execution in a clear and concise manner. The ultimate goal of ARAN-LANG is to restore the core simplicity of JavaScript, but it is constrained by the other two criteria.

JavaScript Subset: ARAN-LANG should be reasonably close to a subset of JavaScript so that ARAN's API remains understandable to JavaScript developers. We do not want developers to have to study this intermediate language to be able to properly use our approach.

Expressive: ARAN-LANG should be sufficiently expressive to support the full ECMAScript specification without requiring excessive additional logic. Indeed, while additional logic is necessary to express complex features using simpler ones, it should be kept to a minimum to reduce code bloat and performance overhead. Also, this additional logic will look unfamiliar to the end user of the analysis and could make the results of the analysis hard to interpret.

Being an intermediate compilation language, ARAN-LANG does not require an actual syntax. Instead, in Appendix B, we present the AST format of ARAN-LANG as a TypeScript type definition. In the remainder of this section, we discuss several points of interest regarding the language.

Strict Mode Only JavaScript can be executed in two modes: *strict* mode and *sloppy* mode. In strict mode, some errors are thrown instead of being ignored, and some features, such as the WITH statement are disabled. These features are often considered legacy and preclude JavaScript runtimes from carrying out important optimizations. We made the design decision to always output instrumented code in strict mode. While this completely hides the complexity of dealing with two modes from the user, it forces us to remove all the sloppy-only features of JavaScript from ARAN-LANG.

Intrinsic Record Often, logic added by ARAN involves accessing pre-existing values from the global object. These values are usually referred to as *intrinsic* values. For instance, the regexp literal `/pattern/u` can be expressed as `new·RegExp("pattern","u")`. In doing so, we have to ensure that the variable `RegExp` still refers to the intrinsic `%RegExp%`. In ARAN-LANG, intrinsic values can be retrieved by name with `IntrinsicExpression`. This requires executing some setup code to load all the intrinsic values of interest to ARAN in a safe location. Additionally, ARAN defines some custom values in that location – e.g., `%aran.binary%` for representing binary operations.

Effect Node In JavaScript, writing to a variable is an expression whose result is the new value of the variable. However, this value is often not used and is simply removed from the value stack – e.g., writing to a variable in a statement context. Hence, two join points are required to implement shadow execution: one for updating the environment and another for removing the value from the stack. Instead, we decided to introduce effect nodes, which are similar to expression nodes but do not produce a value. This way, write operations can be represented by a single join point that always consumes the new value from the value stack. Writes in an expression context are handled with a compilation variable that holds the new value of the variable.

No Imported Variable In a JavaScript module, values from external modules are imported as local variables. These variables are one-way bound to the external value – i.e., they are locally immutable, but mutations from within their source module are reflected. Hence, they can be viewed as syntactic sugar for their external location and be removed from the local scope. In ARAN-LANG, this is made explicit by `ImportExpression`, which contains a `Source` and a `Specifier`.

No Exported Variable In a JavaScript module, local variables can be exported, and any modification to those variables will be visible from other modules. In ARAN-LANG, this coupling is made explicit through an `ExportEffect` which contains a `Specifier` for the name of the export and an `Expression` for the new value of the export. Note that export operations are made into effects for the same reason as write operations.

Explicit Variable Hoisting In JavaScript, variables are actually declared earlier than where their declaration appears. This process is referred to as *hoisting* and is a common source of confusion. In ARAN-LANG, variables are always declared at the beginning of the block where they are hoisted. This translates in ARAN-LANG by the absence of declaration statements and the presence of a list of bindings in `RoutineBlock` and `ControlBlock`. An important behavior that we had to recreate is the temporal deadzone. This timing window spans the moment from where a variable is hoisted to where the declaration statement actually resides. Accessing variables in this temporal deadzone should result in a run-time error, which we implemented through run-time checks. Note that variables in ARAN-LANG have no kind, and that their immutability is also enforced with run-time checks.

Static Variable Access In JavaScript, different types of dynamic frames can reside in a scope. The global object which is at the root every scope and the global declarative record which sits just after the global object are both important instances of dynamic frame. However, the `with` construct and direct calls to `%eval%` can also introduce dynamic frames. In ARAN-LANG, we decided to make the scope static by reifying dynamic frames. Combined with immutability checks and deadzone checks, this means that read and write operations cannot throw exceptions and cannot trigger arbitrary code. This is an important property of ARAN-LANG because it enables the analysis to not check for the result of these operations.

Hard-Coded Parameter Set For the sake of simplicity, ARAN-LANG does not allow for parameter renaming. Instead, all parameters have a fixed name. Some of these parameters already exist in JavaScript – e.g., `this`, `new.target`, and `import.meta`. Some other parameters had to be introduced – e.g., `function.callee`, `function.arguments`, and `catch.error`. We introduced `function.arguments` to replace the traditional `arguments` parameter for two reasons: it is sometimes missing, and it is plagued by legacy features – e.g., 'arguments.callee' and index binding with parameters. Finally, some parameters are functions that are introduced to simplify the language and its associated weaving API – e.g., `import` for representing dynamic import expressions. However, this is not a silver bullet because the analysis must still reason about the meaning of these functions.

Explicit `this` Argument JavaScript implements object-oriented programming by passing a hidden argument that is assigned to the `this` parameter. In ARAN-LANG, the `this` argument is explicitly provided to `ApplyExpression`. Thus, `xs.map(f)` becomes something like `apply((_this_=xs).map,_this_,[f])` [1].

Mandatory Result Value In JavaScript, programs complete with a value based on their last value statement, and functions return a value provided by an optional `return` statement. Because this behavior is hard to reason about, ARAN-LANG represents the body of these two constructs as a `RoutineBlock`, which is parameterized by an expression that is evaluated last and provides the result value of the program or the function. `break` statements are used to model the control flow of RETURN statements.

[1]To eliminate the compilation variable, we explored the idea of introducing a dedicated invoke expression into the language. That would have translated into something like `invoke(xs, "map", [f])`. Unfortunately, this does not work in the presence of side effects because the method must be fetched from the object before evaluating the arguments.

Preserved Generator Head In JavaScript, the code in the parameters of functions can be arbitrarily complex. Our parameter system allowed us to move that logic into the body of functions, which simplifies analysis. Unfortunately, we were not able to apply this simplification for generator functions. This is because the code in the parameters of generator functions is not executed at the same time as the code in its body [2]. Consequently, we had to add an array of `Effect` nodes into `RoutineBlock` to store the head of generator functions.

4. ARAN's Aspect-Oriented API

The main parameter of the weaving process is the pointcut, which can be seen as a predicate for deciding which join points should be intercepted. There is another important option that dictates whether the instrumented code should access the built-in original global declarative record or use a plain object that reifies it. To preserve the safety of variable access, if the built-in global declarative record is used, global variables will be accessed with custom intrinsic functions such as `%aran.readGlobal%`. These functions become unnecessary if the global declarative record is emulated. This is an advantage because the advice will not have to reason about these ARAN-specific intrinsic functions. However, emulating the global declarative record requires instrumenting every single bit of code from the target program to ensure the emulation is not bypassed. As selective instrumentation is an important feature, we envisioned both options.

Interestingly, the logic of the analysis is not directly woven into the target program. Instead, the advice is called by the instrumented code and is expected to adhere to the advice interface presented in Appendix A. Because ARAN-LANG was designed to enable shadow execution, this interface is almost a one-to-one mapping with the ARAN-LANG AST format. The only weaving that is not straightforward is related to blocks:

- First, analyses should be able to reason about the scope and should receive a reified frame upon entering each block. This is provided as a mapping from local variables and parameters to initial values. Variables declared with the `var` and `function` keywords are initialized with the `%undefined%` intrinsic while variables declared through the `let` and `const` keywords are initialized with the `%aran.deadzone%` intrinsic. Reified block frames are provided to both `block@declaration` and `block@declaration-overwrite`. These two join points are similar, but the latter enables the analysis to overwrite the initial value of a variable which is useful for analysis based on wrappers.

- Second, analyses should be able to reason about control flow. To that end, we provide the `block@throwing` join point which is triggered when the block throws an exception. And, we provide the `block@teardown` join point which is triggered upon exiting the block regardless of how it terminated. If either of these join points is intercepted, the block will have to be wrapped inside a `try` statement.

To facilitate state management, every single advice receives a local state as the first argument. This local state can be updated upon entering a block with the `block@setup` advice. This makes it easy to implement list-like data structures and mirror the scope of the program. Local states are also useful for restoring the context of the analysis after a hiatus in the control flow caused by `yield` and `await` expressions.

To locate the AST node that triggered the advice, every single advice receives the path of its triggering node as the last argument. More precisely, this argument is a string that represents the chain of properties leading to the triggering node – e.g., `"$.body.0.expression"`. Other approaches, such as JALANGI [2], use indices instead. While indices should be faster in theory, they also make the overall interface more complex. We decided this was not worth the performance improvement which is likely to be minor thanks to optimizations such as string interning.

[2]We explored the idea of also moving the head of generator functions into their body, but decided against it as we observed too many failures during validation against TEST262.

```
type Variable = string;  type Parameter = string;
type Intrinsic = string; type Path = string;
type Primitive = number | string | boolean | null | { bigint: string };
type TreeLeaf =
  | { type: "primitive"; primitive: Primitive; path: Path; }
  | { type: "closure", kind: string, path: Path }
  | { type: "intrinsic", name: Intrinsic, path: Path }
  | { type: "initial", variable: Variable | Parameter, path: Path }
  | { type: "import", source: string, specifier: string | null, path: Path }
  | { type: "resume", path: Path };
type TreeNode =
  | { type: "apply", function: Tree, this: Tree; arguments: Tree[]; path: Path }
  | { type: "construct", function: Tree, this: Tree[], arguments: Path }
  | { type: "arguments", members: Tree[], path: Path; }
type Tree = TreeLeaf | TreeNode;
type ShadowState = {
  parent: ShadowState | null;
  frame: { [key in Variable | Parameter]?: Tree };
  stack: Tree[];
};
```

Listing 2: The shadow state of our TRACK-ORIGIN analysis.

To evaluate the expressiveness of our approach, we implemented an analysis called TRACK-ORIGIN that records the execution tree of run-time values [3]. This analysis can be leveraged to carry out analyses based on symbolic execution such as concolic testing. Our analysis consists of approximately 300 LoC and can handle all the corner cases of JavaScript. Listing 2 defines the state of the analysis as a TypeScript type definition that mirrors both the value stack and the scope. The code of the analysis is clear and concise; the only logic that we needed to implement involved the transient tracking of shadow values before and after calls to instrumented functions.

It remains an open question whether our API is suitable for implementing analyses truly on a per-project basis. In the negative, we could introduce a domain-specific language for defining the analyses. Alternatively, we could provide a suite of JavaScript APIs, each designed for a specific class of heavyweight dynamic analysis.

5. Evaluating ARAN Against TEST262

To evaluate the reliability of our approach, we implemented it in a tool [4] [5] and tested it against TEST262 [6], which is the official conformance test suite of ECMAScript. It contains about 50,000 test cases that cover the entire ECMAScript specification and even upcoming proposals. We conducted our experiment on the setup described in Table 1. It consists of applying increasingly complex instrumentation to TEST262 cases. At every stage, we excluded the test cases that failed in the previous stages. This helped us to triage failures and diagnose bugs. We describe below our six instrumentation stages [7]; each based on the online architecture depicted in Figure 2.

1. ENGINE: Leaves test cases intact. This stage is intended to uncover the technical limitations of either the underlying NODE runtime or our custom test runner. It also provides a basis to compute the slowdown factors for subsequent stages.

[3]https://github.com/lachrist/aran/blob/664f0a30/test/aspects/track-origin.mjs
[4]https://github.com/lachrist/aran
[5]https://www.npmjs.com/package/aran
[6]https://github.com/tc39/test262
[7]https://github.com/lachrist/aran/tree/664f0a30/test/262/stages
[8]https://github.com/tc39/test262/tree/18ebac8122117bdc55a0d4bba972ba80c0194b41
[9]https://github.com/lachrist/aran/tree/664f0a304b555bcb106f24e72734ad8c88dac429
[10]https://github.com/acornjs/acorn/releases/tag/8.12.1
[11]https://github.com/davidbonnet/astring/releases/tag/v1.9.0

Hardware	Apple Air M2, 2022
NODE	Version v22.3.0
TEST262	Commit SHA: 18ebac81 [8]
ARAN	Commit SHA: 664f0a30 [9]
ACORN	Version 8.12.1 [10]
ASTRING	Version 1.9.0 [11]

Table 1
Setup for testing ARAN against TEST262

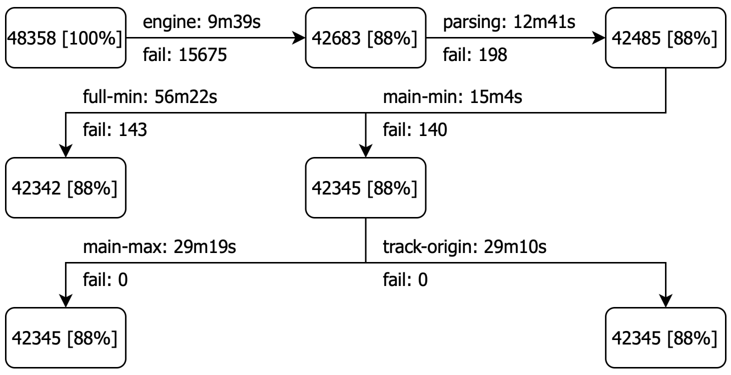

Figure 3: Overview of testing ARAN against TEST262 in 6 stages.

2. PARSING: Parses and directly re-generates static code without manipulating syntactic nodes. This stage is intended to encover the technical limitations either the ACORN parser or the ASTRING generator which are relied upon in the subsequent stages. It also provides insight into the composition of the instrumentation overhead.

3. MAIN-MIN: Applies minimal ARAN instrumentation to the main file of each test case. Test fixtures, module dependencies, and dynamic global code are not instrumented. However, dynamic local code must be instrumented at the `eval@before` join point which is the only pointcut of this stage. Note that even though weaving is minimal, the source code undergoes many during its transformation from JavaScript to ARAN-LANG.

4. FULL-MIN: Still applies minimal ARAN instrumentation, but on every single bit of the code of each test case. This provides us with the opportunity to test the emulation of the global declarative record. This requires advising not only `eval@before` but also `apply@around` and `construct@around` to intercept and instrument dynamic global code provided to the `%eval%` intrinsic and `%Function%` intrinsic. Note that this simple access control system does not intercept indirect applications of intrinsic functions – e.g., `eval.call("code")`.

5. PART-MAX: Applies maximal ARAN instrumentation to the main file of each test case. In this stage, every join point is advised by a function that contains no logic. – e.g., `apply@around` is advised by `(ste,fct,ths,args)=>Reflect.apply(fct,ths,args)`.

6. TRACK-ORIGIN: Our analysis for recording the execution tree of run-time values as presented in Section 4. It is applied to the main file of each test case.

Figure 3 depicts an overview of the results of our experiment. (i) Stage ENGINE introduces about 15k failures, which corresponds to a failure rate of about 12%; by far the highest. Most of these failures were due to features that have not yet been implemented in NODE such as the new `Temporal` API, which accounts for more than 4k failures. Other failures were due to our test runner not entirely adhering to the interface prescribed by TEST262 [12]. For instance, some parts of this interface relate to inter-realm

[12]https://github.com/tc39/test262/blob/main/INTERPRETING.md

communication which occurs via built-in function calls. Whereas, we are interested in preserving the semantics of syntactic constructs. We also discovered actual bugs in NODE, and we reported some of them [13] [14] [15]. (ii) Stage PARSING introduces 198 failures, which were mostly due to ACORN not correctly handling corner cases. (iii) The failures introduced by subsequent stages which actually deploy ARAN are limited in number (143). They provide a good overview of the limitations of our tool as they highlight the semantics that we were not able to preserve. We discuss these discrepancies below and their incidence on the observed failure count.

- *Missing iterable return in pattern* ($42 \Rightarrow 29\%$): Array destructuring assignments – e.g., `[x0,x1]=xs` – are carried out via the iterable protocol [16]. It prescribes calling the `return` method of the iterator when exiting the destructuring assignment; regardless of its outcome. However, after ARAN instrumentation, this method will not be called if an exception is thrown during the iteration. This semantics is challenging to restore because destructuring assignments occur in an expression context and cannot be directly wrapped in a `try` statement.e

- *Corrupt code reification* ($34 \Rightarrow 24\%$): There exist two ways for JavaScript programs to inspect their own source code: the `stack` property of `Error` instances and the `toString` method of `%Function.prototype%`. After instrumentation by ARAN, these two mechanisms will provide a representation of the instrumentation code instead of the original code. This semantics could be restored with some engineering effort. However, we decided against it because the ECMAScript specification does not actually specify the output format of these two mechanisms. As a result, code reification is not guaranteed to be stable across different JavaScript engines and is primarily used for debugging purposes. One might wonder how this discrepancy accounted for 34 failures since it is not part of the specification. The reason is that two functions with the same source code should cause `%Function.prototype.toString%` to produce the same string. Unfortunately, this is not guaranteed after instrumentation because of the compilation variables introduced by ARAN.

- *No two-way bindings for ARGUMENTS* ($32 \Rightarrow 22\%$): In sloppy mode, functions with a simple list of parameters will have them two-way bound with the `arguments` object. This means that changes to the arguments object are reflected in the values of the parameters and vice versa. In a previous version of our tool, we preserved this behavior by leveraging the `Proxy` API [3]. However, we decided to remove this feature because it caused performance overhead and because dynamic argument binding is considered legacy.

- *Hoisted root declarations* ($20 \Rightarrow 14\%$): To simplify its API, ARAN hoists export declarations and variable declarations to the beginning of the sources. Although the temporal deadzone is enforced inside the source, other sources will be able to bypass it. For modules, this is only an issue in the case of circular dependencies. For scripts, this is only an issue if the script synchronously executes another script that uses its own declared global variables. Additionally, because the value of global `const` declarations is not available, ARAN has to turn global `const` declarations into `let` declarations. Although the immutability of the variables will be enforced in the current source, other sources will be able to bypass it.

- *No dynamic function properties* ($2 \Rightarrow 1\%$): Functions created in sloppy mode contain two properties that change dynamically as the function is being called: `arguments` and `caller`. We decided not to preserve this feature because it is technically challenging and because these properties have been deprecated (although they are still part of the ECMAScript 2024 specification).

[13]https://github.com/nodejs/node/issues/52720
[14]https://github.com/nodejs/node/issues/53575
[15]https://github.com/nodejs/node/issues/52737
[16]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterable_protocol

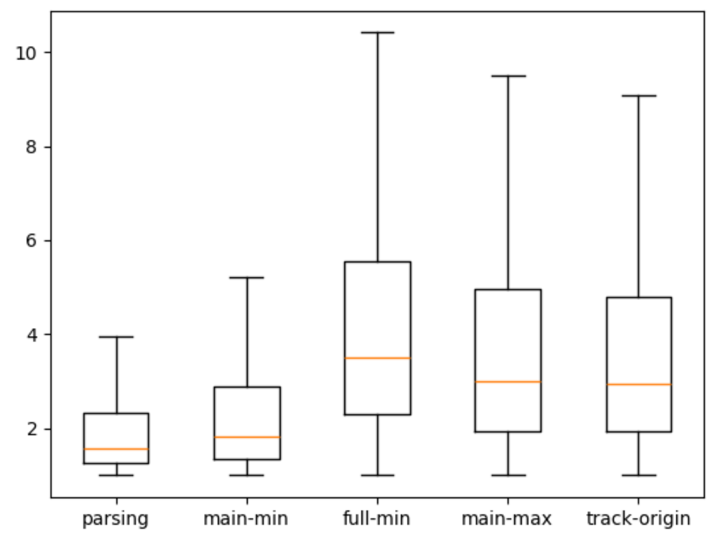

Figure 4: Distribution of the slowdown factor of each stage compared to the calibrating ENGINE stage.

- *Other Discrepancies* $(13 \Rightarrow 9\%)$: We observe other discrepancies that require code that is so convoluted that it is unlikely to cause issues in practice. For instance, after ARAN instrumentation of a derived class, the `prototype` property of the parent class is accessed twice instead of once. This could be observed with a getter or with the `Proxy` API. A detailed list of the discrepancies can be found in the repository of our tool [17].

To prevent flakiness, TEST262 was designed to be independent of performance. However, performance overhead is *the* discrepancy most likely to cause issues in practice. Indeed, as the performance overhead of the analysis increases, so does the chance that time-sensitive applications will be affected. For instance, some event interleaving might be observed by the analysis, whereas such interleaving may never occur without it. To provide insight into the performance overhead of our approach, we recorded the time required to execute each test case in each stage. Figure 4 depicts the distribution of slowdown factors that were observed in each stage compared to the initial ENGINE stage. It appears that slowdown factors rarely exceed 10x, which is corroborated by the total time of each stage shown in Figure 3.

Although not depicted in Figure 3, outliers can reach a slowdown factor of up to 800x. They were removed from the graph because they made it hard to read. We selected the set of pathological test cases that made the slowdown factor of stage PART-MIN exceed 100x. This selection contains 32 test cases, among which 31 were large and probably generated files of several thousands lines of code – e.g.,[18]. This made us suspect that the overhead was largely due to the instrumentation. Figure 5 depicts the distribution of the time spent instrumenting code compared to the total time required to execute each test case. It appears that instrumentation accounts for about half of the time required to execute a typical test case. However, Figure 6 depicts the same data but only for the pathological selection. It appears that instrumentation accounts for almost all the time required to run a pathological test case. This is a positive outcome because instrumentation overhead can be lifted by performing instrumentation statically via the offline architecture depicted in Figure 1.

We are aware that TEST262 is not tailored to benchmark performance. Hence, the 10x slowdown factor we observed is a crude approximation of how our approach could perform in practice. Further work is required to establish the transparency of our approach for real-world time-sensitive applications.

[17]https://github.com/lachrist/aran/blob/664f0a30/doc/issues/missing-iterable-return-call-in-pattern.md
[18]https://github.com/tc39/test262/blob/867ca540/test/language/identifiers/start-unicode-10.0.0-class-escaped.js

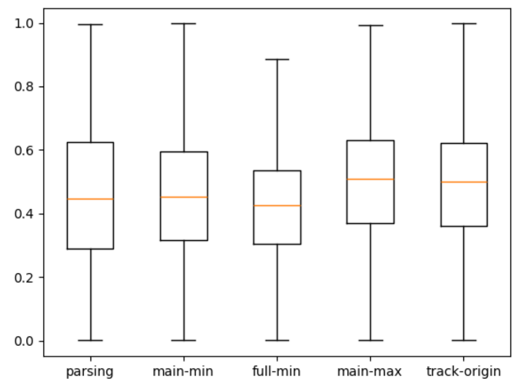

Figure 5: Distribution of instrumentation time ratio of the entire test suite.

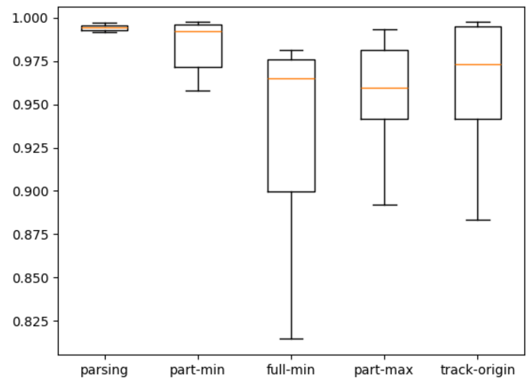

Figure 6: Distribution of instrumentation time ratio of pathological test cases.

6. Related Work

Driven by the popularity and dynamic nature of JavaScript, the research community has proposed a wealth of approaches for dynamically analyzing JavaScript programs. By far, the most successful class of dynamic program analysis for JavaScript is the so-called "lightweight" dynamic analyses such as: code coverage [19], profiling [4, 5] [20], and dynamic linting [6, 7, 8]. In this section, we briefly review the state-of-the-art approaches that are capable of supporting shadow execution of JavaScript and implementing so-called "heavyweight" dynamic analyses.

Linvail Most importantly, the work we presented in this paper iterates on previous work presented in [9]. While this earlier work also relies on source code instrumentation to carry out shadow execution, it focuses on an interesting technique called *membrane*, which involves using the `Proxy` JavaScript API [3] to enforce an access control system between instrumented and non-instrumented code areas. This membrane was leveraged to track primitive values for longer periods of time. This approach was implemented in a library [21] which can still be integrated into our approach. This paper proposes several novelties. First, we came up with an aspect-oriented API to define the logic of the analysis which is more flexible than our previous API. Second, we crafted an intermediate language that strikes an interesting balance between expressiveness and simplicity. Third, we confronted our approach against the complexity of the ECMAScript specification.

Jalangi Apart from our own work, the closest related work is Jalangi [2]. It is another approach for building heavyweight dynamic analyses through JavaScript source code instrumentation. It is however unclear how Jalangi can handle the current complexity of JavaScript. While the first version of Jalangi [22] has been archived, the second version of Jalangi [23] is still maintained. However, is has not kept pace with ECMAScript's fast release cycle as it only supports ECMAScript 5.1 [24] which dates back to June 2011.

[19]https://github.com/bcoe/c8
[20]https://www.dynatrace.com
[21]https://github.com/lachrist/linvail
[22]https://github.com/SRA-SiliconValley/jalangi
[23]https://github.com/Samsung/jalangi2
[24]https://github.com/Samsung/jalangi2#supported-ecmascript-versions

Dynamic Taint Analysis A prime application of shadow execution is *dynamic taint analysis* which enforces security policies at run-time by propagating security-related labels along with run-time values. As security is an important concern for JavaScript programs, multiple taint analysis approaches have been proposed [10, 11, 12]. Most of these approaches rely on a modified runtime to carry out the analysis. While runtime instrumentation is attractive for conducting experiments, we believe it is not suitable for wide adoption. In contrast, our tool is based on source code instrumentation and is designed to be deployed on any JavaScript runtime. To the best of our knowledge only ICHNAEA [12] is based on source code instrumentation. There are two major differences between their work and ours. First, they provide an API that forces the user into a specific shadow execution model optimized for taint analysis, whereas we provide a flexible aspect-oriented API. Second, similarly to JALANGI, ICHNAEA only supports ECMAScript 5.1.

Dynamic Symbolic Execution Another prominent application of shadow execution is *dynamic symbolic execution* which attempts to provide inputs to a program that will lead it to a specific execution path by labeling run-time values with symbols. It is often used to generate test inputs, an approach known as *concolic testing* [13, 14]. Multiple dynamic symbolic execution frameworks have been proposed for JavaScript [15, 16, 17]. Similar to dynamic taint analysis, these approaches often rely on a modified runtime, whereas we explored source code instrumentation for applicability reasons. The only symbolic execution framework based on source code instrumentation that we are aware of is KUDZU [15], which relies on JALANGI instrumentation and suffers from its limitations.

Value Virtualization There has been some work on virtualizing values in JavaScript which could provide the basis for shadow execution [18, 19, 20, 21]. However, these approaches require executing JavaScript code in a virtual machine, which may limit their applicability and explain why they have not been adopted in practice. Actually, ECMAScript provides its own standard virtualization API called `Proxy` [3]. Unfortunately, this API focuses on object values and is cannot virtualize primitive values. Despite shown interest [25], there is little chance that the `Proxy` API will be extended to primitive values due to performance implications.

Low-Level Shadow Execution There is also an interesting body of work on performing shadow execution on the lower-level representation of programs. For instance, VALGRIND [22] is a popular framework for instrumenting binaries for the purpose of dynamic analysis and is fully capable of carrying out shadow execution. However, because JavaScript is primarily interpreted, and although it is sometimes JIT-compiled, it is unclear how this approach could be applied to a wide range of rapidly evolving runtimes.

7. Conclusion

We introduced ARAN, an approach for implementing heavyweight dynamic program analyses. Our method is broadly applicable, relying exclusively on source code instrumentation. However, further research is needed to explore deployment infrastructures that do not compromise this applicability. Our approach is also expressive, featuring an aspect-oriented API that facilitates shadow execution of complex JavaScript programs with just 31 entry points. Furthermore, it maintains transparency by preserving the semantics of the analyzed program, achieving a 99.7% success rate on TEST262. Future work should also examine whether the performance overhead remains acceptable in real-world applications.

[25]https://github.com/hugoattal/tc39-proposal-primitive-proxy

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11, Springer, 1997, pp. 220–242.

[2] K. Sen, S. Kalasapur, T. Brutch, S. Gibbs, Jalangi: A selective record-replay and dynamic analysis framework for javascript, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 488–498.

[3] T. Van Cutsem, M. S. Miller, Proxies: design principles for robust object-oriented intercession apis, ACM Sigplan Notices 45 (2010) 59–72.

[4] S. H. Jensen, M. Sridharan, K. Sen, S. Chandra, Meminsight: platform-independent memory debugging for javascript, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 345–356.

[5] L. Gong, M. Pradel, K. Sen, Jitprof: Pinpointing jit-unfriendly javascript code, in: Proceedings of the 2015 10th joint meeting on foundations of software engineering, 2015, pp. 357–368.

[6] A. M. Fard, A. Mesbah, Jsnose: Detecting javascript code smells, in: 2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2013, pp. 116–125.

[7] L. Gong, M. Pradel, M. Sridharan, K. Sen, Dlint: Dynamically checking bad coding practices in javascript, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 94–105.

[8] M. Pradel, P. Schuh, K. Sen, Typedevil: Dynamic type inconsistency analysis for javascript, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, IEEE, 2015, pp. 314–324.

[9] L. Christophe, E. G. Boix, W. De Meuter, C. De Roover, Linvail: A general-purpose platform for shadow execution of javascript, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, 2016, pp. 260–270. doi:10.1109/SANER.2016.91.

[10] S. Wei, B. G. Ryder, Practical blended taint analysis for javascript, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, 2013, pp. 336–346.

[11] B. Stock, S. Lekies, T. Mueller, P. Spiegel, M. Johns, Precise client-side protection against dom-based cross-site scripting, in: 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 655–670.

[12] R. Karim, F. Tip, A. Sochůrková, K. Sen, Platform-independent dynamic taint analysis for javascript, IEEE Transactions on Software Engineering 46 (2018) 1364–1379.

[13] P. Godefroid, N. Klarlund, K. Sen, Dart: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp. 213–223.

[14] K. Sen, Concolic testing, in: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering, 2007, pp. 571–572.

[15] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, A symbolic execution framework for javascript, in: 2010 IEEE Symposium on Security and Privacy, IEEE, 2010, pp. 513–528.

[16] B. Loring, D. Mitchell, J. Kinder, Expose: practical symbolic execution of standalone javascript, in: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, 2017, pp. 196–199.

[17] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, P. Gardner, Symbolic execution for javascript, in: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, 2018, pp. 1–14.

[18] Y. Cao, Z. Li, V. Rastogi, Y. Chen, Virtual browser: a web-level sandbox to secure third-party javascript without sacrificing functionality, in: Proceedings of the 17th ACM conference on Computer and communications security, 2010, pp. 654–656.

[19] T. H. Austin, T. Disney, C. Flanagan, Virtual values for language extension, in: Proceedings of the 26th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA11), 2011, pp. 921–938.

[20] T. Kataoka, T. Ugawa, H. Iwasaki, A framework for constructing javascript virtual machines with customized datatype representations, in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, 2018, pp. 1238–1247.
[21] T. Ugawa, H. Iwasaki, T. Kataoka, ejstk: Building javascript virtual machines with customized datatypes for embedded systems, Journal of Computer Languages 51 (2019) 261–279.
[22] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proceedings of the 2007 Conference on Programming Language Design and Implementation (PLDI07), 2007, pp. 89–100.

A. ARAN's Advice API

```
import type { Header } from "./lang";
import type { DeepLocalSitu } from "./situ";

type StackValue = unknown; type ScopeValue = unknown; type OtherValue = unknown;
type Source = string;      type Specifier = string;   type Delegate = boolean;
type Intrinsic = string;   type Label = string;       type Path = string;
type Variable = string;    type Parameter = string;

type ProgramKind = "module" | "script" | "global-eval" | "root-local-eval" | "deep-local-eval";
type ArrowKind = "arrow" | "async-arrow";
type FunctionKind = "function" | "async-function";
type GeneratorKind = "generator" | "async-generator";
type MethodKind = "method" | "async-method";
type ClosureKind = ArrowKind | FunctionKind | GeneratorKind | MethodKind;
type ControlKind = "try" | "catch" | "finally" | "then" | "else" | "while" | "bare";
type BlockKind = ProgramKind | ClosureKind | ControlKind;
type TestKind = "if" | "while" | "conditional";

type Key = Parameter | Variable;
type Frame = { [key in Key]?: ScopeValue };
type Primitive = string | number | bigint | null | boolean;

type Advice<S> = {
  "block@setup": (s:S, k:BlockKind, p:Path) => S;
  "program-block@before": (s:S, k:ProgramKind, hs:Header[], p:Path) => void;
  "closure-block@before": (s:S, k:ClosureKind, p:Path) => void;
  "control-block@before": (s:S, k:ControlKind, ls:Label[], p:Path) => void;
  "block@declaration": (s:S, k:BlockKind, f:Frame, p:Path) => void;
  "block@declaration-overwrite": (s:S, k:BlockKind, f:Frame, p:Path) => Frame;
  "generator-block@suspension": (s:S, k:GeneratorKind, p:Path) => void;
  "generator-block@resumption": (s:S, k:GeneratorKind, p:Path) => void;
  "program-block@after": (s:S, k:ProgramKind, v:StackValue, p:Path) => OtherValue;
  "closure-block@after": (s:S, k:ClosureKind, v:StackValue, p:Path) => OtherValue;
  "control-block@after": (s:S, k:ControlKind, p:Path) => void;
  "block@throwing": (s:S, k:BlockKind, v:OtherValue, p:Path) => void;
  "block@teardown": (s:S, k:BlockKind, p:Path) => void;
  "break@before": (s:S, l:Label, p:Path) => void;
  "test@before": (s:S, k:TestKind, v:StackValue, p:Path) => boolean;
  "intrinsic@after": (s:S, i:Intrinsic, v:OtherValue, p:Path) => StackValue;
  "primitive@after": (s:S, v:Primitive, p:Path) => StackValue;
  "import@after": (s:S, r:Source, k:Specifier|null, v:OtherValue, p:Path) => StackValue;
  "closure@after": (s:S, k:ClosureKind, f:Function, p:Path) => StackValue;
  "read@after": (s:S, k:Key, v:ScopeValue, p:Path) => StackValue;
  "eval@before": (s:S, c:DeepLocalSitu, v:StackValue, p:Path) => StackValue|OtherValue;
  "eval@after": (s:S, v:OtherValue, p:Path) => StackValue;
  "await@before": (s:S, v:StackValue, p:Path) => OtherValue;
  "await@after": (s:S, v:OtherValue, p:Path) => StackValue;
  "yield@before": (s:S, d:Delegate, v:StackValue, p:Path) => OtherValue;
  "yield@after": (s:S, d:Delegate, v:OtherValue, p:Path) => StackValue;
  "drop@before": (s:S, v:StackValue, p:Path) => OtherValue;
  "export@before": (s:S, k:Specifier, v:StackValue, p:Path) => OtherValue;
  "write@before": (s:S, k:Key, v:StackValue, p:Path) => ScopeValue;
  "apply@around": (s:S, f:StackValue, t:StackValue, xs:StackValue[], p:Path) => StackValue;
  "construct@around": (s:S, f:StackValue, xs:StackValue[], p:Path) =>StackValue;
};
```

B. ARAN's Intermediate Language

```
// Brand //

type Source = string;
type Specifier = string;
type Variable = string;
type Label = string;

// Header //

type DeclareHeader = {
  type: "declare";
  kind: "let" | "var";
  variable: Variable;
};

type ImportHeader = {
  type: "import";
  source: Source;
  import: Specifier | null;
};

type ExportHeader = {
  type: "export";
  export: Specifier;
};

type AggregateHeader = {
  type: "aggregate";
  source: Source;
  import: Specifier | null;
  export: Specifier;
} | {
  type: "aggregate";
  source: Source;
  import: null;
  export: null;
};

type ModuleHeader = ImportHeader | ExportHeader | AggregateHeader;

// Intrinsic //

// For brevity, we only list a couple of regular intrinsics.
type RegularIntrinsic =
    | "undefined"
    | "eval"
    | "String"
    | "RegExp"
    | "TypeError"
    | "ReferenceError"
    | "SyntaxError"
    | "Symbol.unscopables"
    | "Symbol.asyncIterator"
    | "Symbol.iterator"
    | "Reflect.get"
    | "Reflect.has"
    | "Reflect.set";

type AccessorIntrinsic =
    | "Symbol.prototype.description@get"
    | "Function.prototype.arguments@get"
    | "Function.prototype.arguments@set";

type AranIntrinsic =
    | "aran.global"
    | "aran.declareGlobal"
    | "aran.readGlobal"
    | "aran.typeofGlobal"
    | "aran.discardGlobal"
    | "aran.writeGlobalStrict"
    | "aran.writeGlobalSloppy"
    | "aran.record"
    | "aran.unary"
    | "aran.binary"
    | "aran.throw"
    | "aran.get"
    | "aran.deadzone"
    | "aran.toPropertyKey"
    | "aran.isConstructor"
    | "aran.toArgumentList"
    | "aran.sliceObject"
    | "aran.listForInKey"
    | "aran.listRest"
    | "aran.createObject"
    | "aran.AsyncGeneratorFunction.prototype.prototype"
    | "aran.GeneratorFunction.prototype.prototype";

  type Intrinsic = RegularIntrinsic | AccessorIntrinsic | AranIntrinsic;

  // Parameter //

  type GlobalProgramParameter = "import" | "import.meta" | "this";

  type LocalProgramParameter =
    | "super.get"
    | "super.set"
    | "super.call"
    | "private.check"
    | "private.get"
    | "private.has"
    | "private.set"
    | "scope.read"
    | "scope.writeStrict"
    | "scope.writeSloppy"
    | "scope.typeof"
    | "scope.discard";

  type ProgramParameter = GlobalProgramParameter | LocalProgramParameter;

  type ClosureParameter =
    | "this"
    | "new.target"
    | "function.arguments"
    | "function.callee";

  type CatchParameter = "catch.error";

  type Parameter = ProgramParameter | ClosureParameter | CatchParameter;

  // Program //

  type Program = {
    type: "Program";
    kind: "module";
    situ: "global";
    head: ModuleHeader[];
    body: RoutineBlock & { head: null };
  } | {
    type: "Program";
    kind: "script";
    situ: "global";
    head: DeclareHeader[];
    body: RoutineBlock & { head: null };
  } | {
    type: "Program";
    kind: "eval";
    situ: "global";
    head: DeclareHeader[];
    body: RoutineBlock & { head: null };
  } | {
    type: "Program";
    kind: "eval";
    situ: "local.root";
    head: DeclareHeader[];
    body: RoutineBlock & { head: null };
  } | {
    type: "Program";
    kind: "eval";
    situ: "local.deep";
    head: [];
    body: RoutineBlock & { head: null };
  };

  // Block //

  type ControlBlock = {
    type: "ControlBlock";
    labels: Label;
    bindings: [Variable, Intrinsic][];
    body: Statement[];
  };

  type RoutineBlock = {
    type: "RoutineBlock";
    bindings: [Variable, Intrinsic][];
    head: Effect[] | null;
    body: Statement[];
    tail: Expression;
  };

  // Statement //

  type Statement = {
    type: "EffectStatement";
    inner: Effect;
  } | {
    type: "BreakStatement";
    label: Label;
  } | {
    type: "DebuggerStatement";
  } | {
    type: "BlockStatement";
    body: ControlBlock;
  } | {
    type: "IfStatement";
    test: Expression;
    then: ControlBlock;
    else: ControlBlock;
  } | {
    type: "WhileStatement";
    test: Expression;
    body: ControlBlock;
  } | {
    type: "TryStatement";
    try: ControlBlock;
    catch: ControlBlock;
    finally: ControlBlock;
  };

  // Effect //

  type Effect = {
    type: "ExpressionEffect";
    discard: Expression;
  } | {
    type: "ConditionalEffect";
    test: Expression;
    positive: Effect[];
    negative: Effect[];
  } | {
    type: "WriteEffect";
    variable: Parameter | Variable;
    value: Expression;
  } | {
    type: "ExportEffect";
    export: Specifier;
    value: Expression;
  };

```

```
// Expression //

type Expression = {
  type: "PrimitiveExpression";
  primitive: null | boolean | number | string | { bigint: string };
} | {
  type: "IntrinsicExpression";
  intrinsic: Intrinsic;
} | {
  type: "ImportExpression";
  source: Source;
  import: Specifier | null;
} | {
  type: "ReadExpression";
  variable: Parameter | Variable;
} | {
  type: "ClosureExpression";
  kind: "arrow" | "function" | "method";
  asynchronous: boolean;
  body: RoutineBlock & { head: null };
} | {
  type: "ClosureExpression";
  kind: "generator";
  asynchronous: boolean;
  body: RoutineBlock & { head: Effect[] };
} | {
  type: "AwaitExpression";
  promise: Expression;
} | {
  type: "YieldExpression";
  delegate: boolean;
  item: Expression;
} | {
  type: "SequenceExpression";
  head: Effect[];
  tail: Expression;
} | {
  type: "ConditionalExpression";
  test: Expression;
  consequent: Expression;
  alternate: Expression;
} | {
  type: "EvalExpression";
  code: Expression;
} | {
  type: "ApplyExpression";
  callee: Expression;
  this: Expression;
  arguments: Expression[];
} | {
  type: "ConstructExpression";
  callee: Expression;
  arguments: Expression[];
};

```