Efficient Module-Level Dynamic Analysis for Dynamic Languages with Module Recontextualization

Nikos Vasilakis
MIT CSAIL
USA
nikos@vasilak.is

Grigoris Ntousakis
TU Crete
Greece
gntousakis@isc.tuc.gr

Veit Heller
Unaffiliated
Germany
veit@veitheller.de

Martin C. Rinard
MIT CSAIL
USA
rinard@csail.mit.edu

ABSTRACT

Dynamic program analysis is a long-standing technique for obtaining information about program execution. We present *module recontextualization*, a new dynamic analysis approach that targets modern dynamic languages such as JavaScript and Racket, enabled by the fact that they feature a module-import mechanism that loads code at runtime as a string. This approach uses lightweight load-time code transformations that operate on the string representation of the module, as well as the context to which it is about to be bound, to insert developer-provided, analysis-specific code into the module before it is loaded. This code implements the dynamic analysis, enabling this approach to capture all interactions around the module in unmodified production language runtime environments. We implement this approach in two systems targeting the JavaScript and Racket ecosystems. Our evaluation shows that this approach can deliver order-of-magnitude performance improvements over state-of-the-art dynamic analysis systems while supporting a range of analyses, implemented on average in about 100 lines of code.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; **Dynamic analysis**; *Frameworks.*

KEYWORDS

Dynamic, Runtime, Analysis, Instrumentation, Recontextualization, Performance, Security

ACM Reference Format:
Nikos Vasilakis, Grigoris Ntousakis, Veit Heller, and Martin C. Rinard. 2021. Efficient Module-Level Dynamic Analysis for Dynamic Languages with Module Recontextualization. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468574

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468574

1 INTRODUCTION

Dynamic program analysis is a technique for monitoring, understanding, and potentially intervening in program behavior during its execution. To cite only a few examples, dynamic analysis has been used to infer invariants, check security constraints, and extract performance characteristics [3, 29].

Existing dynamic analyses often impose significant runtime overhead—Jalangi [33] and RoadRunner [13], for example, report No-Op analysis overheads on the order of 26–32× and 52×, respectively. For this reason, dynamic analysis is typically deployed for offline use—collecting and replaying traces offline or stressing a program with test inputs in a test environment. The fact that production environments can differ considerably from offline or testing environments can significantly impair the utility of dynamic analyses that are deployed only during development or testing. Software vulnerabilities, for example, can be latent during development and test, but exploitable only in production [30].

We present a new point in the dynamic analysis design space: *module recontextualization* is an approach that operates at the granularity of modules, with the resulting analysis code executing at module boundaries.[1] We emphasize that the goal is not to supplant existing techniques that operate at the granularity of instructions or procedures [22, 26, 33]. The goal is instead to provide a coarse analysis with low enough overhead (in practice, 2–3% runtime overhead) to enable always-on, uniform deployment during development, testing, and production. In effect, we trade off detail and precision to drive down the overhead while still supporting meaningful analyses (§5).

Module recontextualization leverages characteristics of modern dynamic languages to dynamically transform each module when it is loaded, applying both source code and object transformations. It thus requires no changes to the runtime environment and works with completely unmodified dynamic language production runtimes. The analyses themselves are written in the same language as the analyzed program, preserving developer knowledge, expertise, libraries, and code, and enabling the development of analyses that analyze analysis code. These analyses remain fully under developer control, with module recontextualization supporting targeted analysis of only selected modules and dynamic toggling on and off as the application executes.

[1] We use the terms module and library interchangeably.

This paper makes the following contributions:

- **Module recontextualization:** It presents module recontextualization, a new dynamic analysis approach that targets modern dynamic languages and operates at the boundaries of (selected) modules. Module boundaries go well beyond externally invoked entry points—they also include referenced global variables, basic language features such as `import` and `export` statements, and basic type constructors such as `Number` and `Array` constructors.
- **Two implementations:** It presents a two-part implementation of module recontextualization, Lya, that targets the JavaScript and Racket ecosystems. The discussion of the opportunities and challenges associated with implementing Lya as a pluggable library for modern dynamic languages focuses on the JavaScript port, and discusses Racket when the two diverge.
- **Case studies and evaluation:** It presents an evaluation of Lya for three dynamic analyses of JavaScript applications and libraries, including a read/write/execute security analysis, a performance analysis, and a run-time type invariant discovery analysis. It shows that Lya incurs runtime overheads under 5% and can accurately detect issues and application characteristics that would surface only in production environments—*e.g.*, invalid accesses during object deserialization, performance pathologies in regular-expression matching, and dynamic type anomalies.

Lya has been open-sourced and is available for download from GitHub:

https://github.com/andromeda/lya

2 BACKGROUND, EXAMPLES, AND SCOPE

We first present background on module systems employed by dynamic languages (§2.1). We then describe three use cases that highlight the kinds of the analyses the Lya is designed to support (§2.2). We finish by identifying the scope of Lya, *i.e.*, the characteristics of the environments and analyses that it targets (§2.3).

2.1 Module Systems

Modules encapsulate reusable functionality. This functionality typically falls into two categories: it either (i) comes bundled with the language, possibly wrapping operating-system interfaces such as the file system in a way that is system-agnostic and conforms to the language's conventions, or (ii) is provided by other developers sharing code others might find useful. Consider a module named `simple-math` below, providing a few mathematical functions such as `mul` and `div`: [2]

```
let math = {
  mul: (a, b) => a * b,
  div: (a, b) => {
    import("log").info(b);
    return a / b } };
// ...some more code...
exports = math;
```

This module may be `import`ed and used by a different module, as shown below:

```
let m = import("simple-math");
let result = m.div(m.mul(1, 2), m.mul(3, 4));
print(result); exports = result;
```

From a developer's perspective, `import`ing a module makes its functionality available to the calling code by means of binding its functionality to a name in the caller's scope. This is achieved by some form of `export`ing, where the module developer expresses which values should become available to the `import`ing code. The definition of a value depends on the semantics of the language. Internally, the module may `import` other modules, cause side effects to the file system or the network, or even be implemented in multiple languages.

Importing a module in a dynamic language such as JavaScript typically involves several steps. The runtime system first locates the module in the file system. It then reads the module and wraps it to resolve module-local names, such as `__filename` in JavaScript and `__name__` in Python, to meaningful values. The wrapper is then interpreted and evaluated using the language's interpreter, which might result in side effects—for example, a `process.exit()` in the module's top-level scope will exit the entire program. Finally, the value bound to the `export`ed interface or returned from this interpretation (depending on the language) is made available to the scope of the `import`ing code.

Complications may include the use of a module cache to avoid loading overheads and maintain consistency for modules that are loaded multiple times from different parts of the code base. The use of a module cache can also support recursive imports and cyclic dependences. An increasingly common feature is to allow different versions of the same module to co-exist in a program, to avoid imposing one mutually exclusive choice—a paradoxical situation known as "dependency hell". As a result, a single `import` *l* may not necessarily resolve to the same (version of the) module *l* every time. The dual of this is also possible: two different module names may resolve to the same identifier (*i.e.*, point to the same cache entry). These features can significantly complicate dynamic analyses that operate at the granularity of modules. We further discuss these issues, including how Lya deals with them, in Section 3.

2.2 Dynamic Analysis Examples

We next discuss three example dynamic analyses that can be performed at the granularity of modules: (i) a read-write-execute security analysis, (ii) a performance-profiling analysis, and (iii) an analysis extracting runtime type invariants.

Security Analysis: The pervasive reliance on third-party libraries has led to an explosion of supply-chain attacks [18, 20, 23, 35, 39]. Both bugs and malicious code in libraries create attack vectors, exploitable long after libraries reach their end users. Popular libraries, depended upon by tens of thousands of other libraries or applications, can allow vulnerabilities deep in the dependency graph to affect a great number of applications [43, 44].

Consider, for example, the recent `event-stream` incident [11, 30], in which a maintainer of a highly popular library inserted code to steal Bitcoin wallet credentials from programs using that library. Heavyweight testing or instrumentation [33] would not have helped, as `event-stream` activated only during production rather

[2] In the background and design sections of the paper, we write `import`; in the sections describing the two implementations and evaluation of Lya, we use the actual name corresponding to the prototype in each language—*e.g.*, `require`.

than during testing or development. Whole-program OS-level containment or system-call interposition [32], would have not helped either as the programs importing `event-stream` already made use of system calls to access the disk and network. Finally, static analysis would have been of little use, as `event-stream` encrypted its malicious payload.

A module-level dynamic analysis of read/write/execute permissions [40] used by this library would have detected the unusual resources accessed by `event-stream`. Analyzing the behavior *within* the library itself is not critical: if any data exfiltration is happening, it will require calling out of the library and into the network—in `event-stream`'s case, using the `fs` library to modify a different library and then call `http` from the second library. Both `fs` and `http` are part of the standard library, built into the runtime environment. Other examples of interfaces that are available to the entire program include global variables, library importing, and the module cache—all of which are accessible by any third-party library.

Performance Diagnosis: Diagnosing performance problems is a difficult task, exacerbated by the heavy use of third-party libraries. These libraries often work well until there is an unexpected change in the type or characteristics of the workload [38]. In many cases, the performance behavior of these libraries is affected by a single unusual input.

Consider, for example, the `minimatch` library, a regular-expression-based file-matching utility susceptible to long delays due to regular expressions that involve backtracking [21]. Pathological inputs reaching `minimatch`, even if benign, can cause significant performance degradation [6] deep in the dependency chain, affecting also other parts of the program competing for the same resources [7]. Developers use various techniques to understand such problems—*e.g.*, collecting and replaying traces against offline versions of the system, or using statistical profiling to identify hot code-paths. These techniques, however, require significant *manual* effort: capturing traces, setting up test beds, replaying traces, analyzing statistics, and debugging performance are all tedious and time-consuming tasks, compounded by the difficulties of mapping the results to the right third-party libraries.

A library-level profiling analysis would quickly detect any slowdown and appropriately attribute it to the bottlenecked `minimatch`. Wrapping library interfaces with profiling logic can be of aid to constructing a model of the current workload. Such profiling could operate at a high resolution in time and space—at every function call entering a library and on hundreds of libraries across an application—but does not need to track detailed operations such as direct variable accesses. Each library wrapper can collect profiling statistics at its own boundary, aggregating summaries into a global structure ordering libraries by resource consumption.

Type Invariant Discovery: Extracting type information at the module boundary is helpful in a variety of scenarios. For example, it can be used to identify program invariants to be preserved during code modifications [12], or guide program learning and regeneration [4]. Dynamically extracted type information is particularly relevant for dynamically typed languages that have no explicit type information in the language.

Consider, for example, the `gRPC` module for serializing and deserializing objects [37]. To use this module, developers provide a protocol-buffer specification describing the types of values that will be serialized. Given a library—*e.g.*, `bignum`, `crypto`—a developer has to first call it manually, take note of the result's type, and then fill in the protobuf spec. This process has to be repeated with every change, often due to library updates or changes in the consuming program's structure.

Module-level dynamic analysis could be used to discover such type assertions or invariants. The analysis would consult the definition of a type system, capturing the type of values at the boundaries of libraries by observing their arguments during the execution of the program.

2.3 Scope

Lya exploits features of modern dynamic language environments, for example dynamic module loading, runtime metaprogramming facilities such as reflection or exposing object accesses as overloadable functions, and runtime resolution of external references. The basic approach is therefore not appropriate for software written in traditional compiled languages such as C, Java, ML, or Haskell. It is also not appropriate for traditional scripting languages such as the Unix shell due to several challenges [14].

Because Lya operates at the granularity of modules, it targets modern application development methodologies where applications comprise hundreds of modules, with the modules typically reused from large open source repositories such as GitHub or npm. These methodologies deliver applications with (i) a module decomposition coarse enough for minimal runtime overhead, (ii) a module decomposition fine enough to support meaningful analyses that operate at the module granularity (Lya is therefore not well suited for monolithic applications with few or no modules), and (iii) most of the code obtained from external and potentially untrusted sources (motivating the need for dynamic analyses that can pinpoint and help solve security or performance issues).

Our proposed techniques work well when the recency of information (ideally, online) is more important than the level of detail. They meet such recency needs through a combination of factors. First, Lya provides the ability to perform the analysis online by operating at a coarser granularity, by using a production-optimized runtime, and by toggling parts of the analysis on and off. Second, it allows developers to leverage their expertise in their language of choice—rather than introducing a new language only for analysis: the program being analyzed and the program implementing the analysis can only be in the same language, as the analysis transformations are applied dynamically over the program by the same runtime environment. Finally, it deconstructs programs only at library boundaries, a natural boundary for many problems caused (or exacerbated) by third-party libraries.

3 MODULE RECONTEXTUALIZATION

Lya starts by dynamically modifying the functionality of the module system that is responsible for importing and loading modules: instead of simply locating and loading a modules from the file system, the module system yields control to Lya, which applies a series of transformations to modules with the goal of interposing at their boundaries. We start with an overview of Lya (§3.1), highlighting

several key challenges, followed by a detailed description of each step (§3.2–3.4).

3.1 Overview

Lya operates by decomposing the program at the boundaries of modules, applying transformations that insert analysis-specific machinery, and then carefully reassembling individual components to maintain the original semantics:

- **Decomposition:** Lya starts by recursively decomposing a program into its dependencies. This is achieved by rewiring the language's `import` function to go through Lya, resulting in Lya walking the program's recursive dependency structure at runtime. During this phase, Lya has to determine the granularity of the analysis (*e.g.*, top-level modules, a specific module *etc.*) in order to apply transformations at the correct level and map the provided analysis hooks to the corresponding modules.
- **Recontextualization:** Lya then sets up the provided analysis, by transforming each module interface, its surrounding environment, and possibly the values passing through the module boundary. Programmatic transformations walk and wrap each one of these values based on their type. This phase requires solving several challenges, including enumerating all points of entry into and exit out of a module, and swapping all original values externally available to a module with ones that are wrapped with interposition mechanisms.
- **Reassembly** Finally, Lya reassembles individual modified modules back into the program's original structure. A key challenge in this phase is the treatment of the module cache (§2.1), which needs to be augmented to support multiple wrappers per module, each capturing a part of the overall analysis.

Example—Counting Global Accesses: As the three aforementioned analyses (§2.2) are too complex to show here, we present a smaller analysis that counts all accesses to global variables from the `simple-math` module:

```
let fs = import("fs");
let count = {};
forevery.global.inlib(/simple-math/).analyze({
  pre: (name, path, _) => {
    let o = resolve(name, path);
    count[o] = count[o]?  count[o] + 1 : 1;
  } });
process.on("exit", () => { fs.writeFileSync(
    "access.json", "utf-8", JSON.stringify(count)); });
```

Lya-provided `forevery` generates a set of module identifiers. The `inlib` field is a method that takes a regular expression matching module identifiers. If not empty, `pre` and `post` hooks are called before and after each access of the elements specified in the set. Finally, `resolve` is a method for traversing an object given a path within that object. Upon program exit, the results are written to disk, all using the expected Node APIs.

To perform this analysis, Lya first interposes on the `import` call to detect when `simple-math` is loaded. When loading `simple-math`, Lya applies (1) a source-to-source program transformation that redefine global identifiers as module-local ones, and (2) a dynamic metaprogramming (*i.e.*, runtime reflection) transformation to traverse global values and create a global-indirection map specific to the `simple-math` module. For every global identifier the map holds modified global values that are wrapped such that any access to these values from within the `simple-math` module is visible by Lya, which upon access calls the corresponding `pre` hook. Finally, Lya interprets the transformed `simple-math` module using the built-in code evaluation primitive—similar to the vanilla module system—effectively linking the module-local identifier lookups to the map entries that hold the modified values corresponding to these identifiers.

The next few subsections discuss the details.

3.2 Decomposition

When a Lya-augmented program starts, it first loads the analysis file provided by the developer. The file may specify a subset of libraries whose boundaries are of interest or a subset of libraries that should *not* be analyzed. Among other things, Lya needs to determine the library boundaries of interest and the granularity of analysis. To do this, it extracts an approximation of the dependency graph by traversing library files. Using this graph, it processes the analysis file to extract a mapping from library identifiers to analysis hooks. It also checks for constructs not associated with libraries—for example, whether the analysis includes global variables, library-local constructs, standard libraries *etc.* Lya then proceeds to dynamically replace the implementation of `import` and launch the program: rather than simply locating and loading a library, calls to `import` now yield to Lya.

For every invocation of `import` Lya checks the cache (§3.4) to determine (i) if the library has already been loaded, and (ii) whether the library has been loaded with the same analysis hooks. If both are determined true, Lya retrieves the cached version of the library and returns the transformed `export`ed value. If the library was loaded with a different analysis—say because there are different analyses applied to different parts of a dependency tree—Lya constructs the appropriate analysis and applies a transformation pass on a cached copy of the unmodified library (§3.4). Otherwise, Lya first invokes the built-in library loader to locate the library.

The process of loading new libraries includes (i) a phase of reading the necessary source files and (ii) a phase of interpreting them, interspersed by applications of transformations (§3.3). Reading files returns a string representation of the code; interpretation uses the language's runtime evaluation primitives to convert the code into an in-memory object.

Some analyses may themselves make use of global variables, libraries, and other analyzable constructs. As these will be part of the same execution context, Lya must note to avoid transforming and wrapping these constructs as part of the analysis. Lya frameworks may also want to add analysis-specific keywords not provided by the language. To achieve this, Lya wraps each analysis hook with a function whose body starts by defining the expected keywords. The precise techniques for achieving this will be made clear in the next section (§3.3), after covering transformations; the key point to remember is that analyses are initially represented as source strings, similar to libraries.

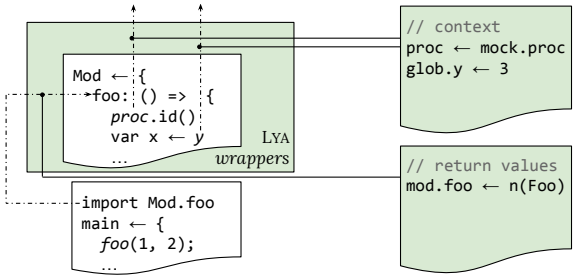

Figure 1: Shadowing segments. Cross-module variable name resolution (left) augmented with Lya (green boxes), which interjects non-bypassable steps resolving to Lya-augmented values (top right: implicit module imports; bottom right: explicit import) (*Cf.*§3.3).

3.3 Recontextualization

For each analyzed library, Lya needs to place hooks all around its boundary—not just its interface entry points (Figure 1). This is achieved in three logical steps: (i) transforming the library's context, a mapping from names to values that are available from outside library, (ii) interpreting the library within this context, so that all names bind and resolve to Lya-augmented values, and (iii) transforming the library's `export` value, *i.e.*, the library interface, once the interpretation is complete. Before discussing *where* each transformation is applied, we show *how* they are applied.

Transformations: Lya's transformations boil down to a base transform `wrap` that traverses and augments values with runtime analysis monitors. At a high level, `wrap` takes a value v and analysis fragments (α_1, α_2) and returns a new value v' that has every one of its fields f wrapped: every f is replaced with a method f' that calls fragment α_1, calls f, calls fragments α_2, and returns the result of the call to f.

More specifically, `wrap` can be applied to any value in the language, which can generally be a primitive, a function, or a compound value—say, a list of values or an object of key-value pairs. Transformations walk compound values from their root, processing component values based on their types (Figure 2): (i) *function* values are wrapped by closures that contain analysis-specific hooks; (ii) *object* and *list* values are recursively transformed, with their getter and setter methods replaced similar to function values; (iii) *primitive* values are either transformed directly or copied unmodified and wrapped with an access interposition mechanism. To avoid cycles during the walk, values are added to a map that is consulted at the beginning of each recursive call.

Direct field accesses, such as assignments, require detection upon access. To achieve this, Lya wraps fields with an interposition mechanism; this mechanism essentially treats direct field accesses as function calls (see §4 for implementation details). Extending a transformed object with a value will start with the value's transformation. For example, if a field in a transformed object is assigned a new value, that value has to be transformed before it is attached to the object.

Lya allows toggling analyses on/off, changing analyses, or chaining multiple analyses during the execution of the program. To achieve this, it maintains a handle to the root of both the unprocessed and the newly processed values, for further processing: the unprocessed value is used to create objects, at runtime, that run different analyses; the new value is used to revoke or chain analyses together.

```
wrap (e: Value, α: Analysis) : Value := match e with
  | {(s, v) :: vs}  →  {(s, wrap v) :: wrap vs}
  | [v :: vs]       →  [(wrap v) :: wrap vs]
  | λ(...args).f    →  λ(...args).{ α1( f( α2(args) ) ) }
  | _               →  interpose(α3, e)
end
```

Figure 2: Base transform. The algorithm (simplified) is presented in functional style to simplify variable binding; types (object, list, function, and primitive), used for pattern matching, are shown in light *turquoise* (*Cf.*§3.3). The functions α_1, α_2, and α_3 stand for the locations of analysis hooks.

Context Transformation: To be able to track an analysis at the library boundary, Lya needs to provide each library with values that are augmented with interposition wrappers—and do this for all of the names to which a library has access. This includes global and pseudo-global[3] names provided by the language and its runtime.

To achieve this, Lya first needs to prepare a transformed copy of the library's context—a map from variable names that are (expected to be) in scope to their values. Lya creates an auxiliary hash table mapping names to transformed values. Names correspond to any name that, by the language definition, is accessible by the library and resolves to a value outside that library, such as globals, built-ins, module-locals, *etc.* Transformed values are created by applying `wrap` to values in the context, adding the provided analysis hooks.

Care must be taken with library-local variables. These are accessible from anywhere within the scope of a library (similar to global variables), but resolve to a different value for each library. Examples include the library's absolute filename as `__name__`, its `export`ed values, and whether the library is invoked as the application's `main` library (§2.1). Attempting to access library-local variables directly from within Lya's scope will fail subtly, as they will end up resolving to library-local values of Lya *itself*—and specifically, the module within Lya that is applying the transformation. Lya solves this problem by leaving the value empty and deferring binding for later from within the scope of the library (see below).

Context Binding: To link the library with the newly transformed version of its context, Lya wraps the library—still an uninterpreted string of source code—with a closure. The closure's body starts with a prologue of the form:

```
local print = ctx.print
local error = ctx.error
// ...more entries...
```

These statements shadow global variable names by redefining them as function-local ones. The closure accepts an argument `ctx` that will hold the customized context (see above), assigning its entries to their respective variable names. The prologue executes before everything else in the library. This technique leverages lexical scoping

[3]For example, Node introduces objects that are not part of the EcmaScript specification into the global scope, such as `process` and `console`; similarly, Lua's Luvit introduces its own globals such as `p()` and `exports`.

to inject a non-bypassable step in the variable name resolution process: instead of resolving to variables in the context, resolution will first "hit" library-local values augmented with analysis monitors.

Late-bound, *library*-local variables, such as the absolute filename mentioned during context creation, are the result of applying `wrap` over variable names in the current scope; these names are now bound to the correct library-local values.

Library Interface Transformation: Returning the library's value to its consumer amounts to interpreting the library, linking it with the custom context, and applying a final transformation to its return value. The goal of the final transformation is to track activity at the boundary.[4] This final transformation is applied for every new consumer of the library, returning a fresh analysis wrapper. This is due to the need for distinguishing between different boundaries of the same library.

The treatment of this feature during reassembly is explained in the next section (§3.4).

3.4 Reassembly

To successfully reassemble the application, Lya needs to ensure that cross-references between libraries resolve correctly. The central mechanism for this resolution is the library cache.

To support multiple wrappers for a single cached library, the cache is extended by two levels (for a total of three). The reason for adding the two levels is that libraries are usually governed by a single context analysis but multiple interface analyses, one for each of their consumers. A context transformation is applied at most a few times (usually only once), whereas a return-value transformation is applied on every **`import`**. Thus, the first level is indexed by library identifiers (as before); the second by context analysis; and the third by analysis of the **`export`**ed interfaces. For each library, the second level contains a collection of entries corresponding to mostly-transformed libraries, and the third level contains fully transformed libraries. Mostly-transformed libraries have gone through the entire transformation pipeline except for the last stage: they have been interpreted and have had their context transformed and linked, but their return value has not been processed to track analysis of its interface.

A special entry is reserved for the original library value as a string (§3.3), so that subsequent transformations can skip loading it from disk. When a new analysis is applied to a library, Lya indexes the cache by library identifier and applies the analysis-specific `wrap` to the library's context. It then adds that result to a slot in the next layer of the cache, indexed by the analysis identifier. When a library is already loaded, Lya indexes by analysis to retrieve the (mostly) transformed library corresponding to this analysis. It then applies a transformation to the library's return value, and inserts the (finalized) transformed library to the third layer of the cache.

4 TWO IMPLEMENTATIONS

We have implemented Lya for server-side JavaScript (Node.js v8.9.4, about 2.5K LoC) and Racket (Racket v7.8, about 500 LoC). This section details the JavaScript implementation, integrated into npm and available for setup under `andromeda/lya`, and only refers to Racket when the two diverge.

There are two main ways to implement Lya. The first is as a modified version of the runtime, in which several stages of its library-loading facility have been augmented in-place. The second approach is to implement Lya as a third-party library (*e.g.*, the `lya` package) available by the language's package manager. With both approaches, the user experience is a backward-compatible, drop-in replacement of the language's module system indistinguishable from the vanilla system. We went with the second approach, as looser coupling seems to have several benefits: it does not force Lya's users to have a custom copy only for running analyses; it removes Lya's developers from the critical path of updates between the language developers and its users; and it simplifies Lya's comparison with the vanilla environment (both in terms of performance and correctness). The primary drawback was missing a few opportunities for lowering runtime performance and development effort.

Module System Implementation: In both languages, the module system is implemented entirely in the respective language itself, exposing a library-local function for importing modules. Loading a fresh module corresponds to the following five stages, all of which are augmented by Lya: (R) Resolution: identify the file to which the module specified corresponds, locate it in the file system, and assign its absolute path as a module identifier. (L) Loading: depending on the file type, identify the corresponding loader—*e.g.*, V8 compiler for `js`, `JSON.parse` for `json` *etc.* (W) Wrapping: wrap the module so that local names do not escape the module's scope and module-local names get resolved. (E) Evaluation: evaluate the wrapped module in the current context, so that global names and top-level objects get resolved correctly. (C) Caching: add the module to a handful of module-related caches, for consistency and performance reasons.

Lya augments all of these steps. Interposing on resolution (R) makes the module identifier available to Lya without affecting the module resolution algorithm. If the module's type corresponds to a module that can be analyzed by Lya, Lya fetches the corresponding analysis during loading (L). Wrapping (W) and evaluation (E) are where Lya transforms the module boundary. Lya adds a wrapper function to pass an additional argument, the modified context `CTX`. Lya for Node comes with a hard-coded list of variable names available to the code of a module, such as `require` and `Array`; the list contains about 150 entries and corresponds to the specific versions of EcmaScript and Node. Identifying names coming from EcmaScript was relatively easy, as the standard makes them explicit. Node's global names are described at various parts of the documentation, but library-local names required close inspection of Node's internals; fortunately, they were only five names.

A challenge with Racket was its lack of facilities for intercepting module loading (L). To address this, Lya provides its own custom loader which allows it to modify the module's source code early—during the macro-expansion phase and prior to any evaluation. During this phase Lya for Racket manipulates the code in AST form, contrary to Lya for JavaScript that manipulates the code as a string. Another challenge with Racket was its lack of support shadowing name bindings inside a module directly when the original value is used. This is because a name is resolved to the name defined in the local module rather than in the parent module, raising

[4] For some analyses, Lya needs to additionally augment the values going through the library's interface—including continuation functions passed as arguments to the library's methods.

a "use before definition". To address this, Lya generates names for the original module in an intermediary module, and binds them to their expected names in the original module. Additionally, if any unused name is not (known to be) bound ahead of time, Racket will complain—which precludes emitting `eval` forms to bind any symbols or introduce any forms. Since the Racket compiler running in phases and the resolution of `require` and macro expansion happen at two distinct phases, Lya shifts phases for the `safe-require` macro to work, expanding to a `module` and `require` form.

We found it useful to add an option for explicitly including and excluding libraries. The configuration object accepts `only` and `not` expressions that indicate whether a module identifier will be part of the analysis. These expressions contain sets of regular expressions, pattern-matching against module identifiers (absolute file-system paths). Originally intended as an aid to Lya's development and debugging, this option proved useful enough for writing the three analyses that we decided to expose it to Lya's users. Examples of its use include excluding built-in libraries or including only the library imported most recently.

Example Transformation Fragments: The code fragments below exemplify Lya's transformations in the context of JavaScript, as applied to the `simple-math` library shown earlier (§2.1).

Lya traverses the `math` object, creating a new replacement object whose functions are replaced with wrappers that call the corresponding function of the original object interleaved with the hooks corresponding to the specific analysis:

```
let _ = math; math = {};
math.mul = (...args) => {
  let p = lya.hooks.prologue(args);
  let v = _.mul(...args);
  return lya.hooks.epilogue(p, v); }
// skipping code for div etc.
```

Lya next creates a modified version of the surrounding context, a binding from names to modified values—*i.e.*, objects transformed according to the aforementioned wrapping transformation:

```
var ctx = {
  // original, unmodified value:
  print:  lya.print,
  // lya-transformed value:
  import: lya.txf(import,
                  lya.hooks.prologue,
                  lya.hooks.epilogue),
  // (...more values below, omitted...)
}
```

Lya binds this new context to the module being loaded using a source-to-source transform that wraps the module in a closure that redefines globally accessible identifiers as module-local ones:

```
function (cxt) {
 var print = cxt.print;
 var import = ctx.import;
  // --start: original math module--
  // (fragments omitted)
  div: (a, b) => {
    log.info.(b);
    return a / b, }
  // --end: original math module--
}
```

It finally interprets the module closure, which returns an in-memory `Function` object, and invokes the closure by passing the modified context created above as an argument.

5 EVALUATION

Questions: We seek to answer the following three questions:

- Q1: How does Lya perform in detecting real problems occurring in production environments?
- Q2: What is the runtime overhead of Lya, and how does it compare with prior analysis frameworks?
- Q3: How large (LoC) are the analyses developed, and how does it compare with prior analysis frameworks?

Summary: We use several different sets of benchmarks to answer these questions. We develop the three analyses outlined in §2 and apply them to both individual libraries and larger programs. The analyses average 94 lines of code, and applied to the tests available by the nominal developers incur runtime overheads of under 5%. We also apply workloads identified from reports taken from GitHub issues and CVE databases and confirm that Lya indeed detects these problematic behaviors. Applying Lya and Jalangi on Jalangi's SunSpider benchmarks shows Lya averaging 87× lower overhead than Jalangi.

Setup: Experiments were conducted on a modest server with 4GB of memory and 2 Intel Core2 Duo E8600 CPUs clocked at 3.33GHz. In terms of software, we used Docker version 18.09.7 running a minimal Ubuntu 14.04.6, Jalangi v1, Node.js v8.9.4 (bundled with V8 v6.1.534.50, LibUV v1.15.0, and npm v6.4.1), and Racket v.7.8—all atop a Debian Linux with kernel v4.4.0-134. All times reported are in *ms*, averaged over 1K runs; *SA*, *PD*, and *TID* respectively stand for security analysis, performance diagnosis, and type invariant discovery.

5.1 Analyses

We now report on the development of the several dynamic analyses, including ones targeting the problems outlined in §2. Individual analyses average 95.3 lines of code, but a significant part (about 20%) of this code is identical across them.

Security Analysis (111LoC): To address the security concerns of third-party libraries, we developed a `RWX` policy that analyzes accesses for every library-to-library combination. The analysis treats built-in libraries and global variables as modules, and develops a permission model where individual fields are governed by permission sets containing combinations of `R`, `W`, and `X` permissions. At the start of the analysis all permissions are set to $\perp$ (*i.e.*, default-deny), and are gradually overwritten based on the accesses seen by the analysis. Example accesses include: (i) reading a value, including assigning it to a variable and passing it around to other modules; (ii) modifying or deleting a value; and (iii) executing a value—*e.g.*, a function or a method—or invoking a constructor (usually prefixed by `new`). The resulting permission sets are organized as collections of maps, one per library, indexed by object paths—*e.g.*, `require("Math").add: RX`.

Table 1: Lya's percent (%) overhead of its analyses applied over 30 popular libraries. On average, Lya's analyses incur an overhead of 4.14% , 3.62%, and 3.86% for the security analysis (SA), performance diagnosis (PD), and type invariant discovery (TID).

	algebra	array.chunk	array-first	array-last	array-range	arr-diff	arr-flatten	concat-stream	deep-bind	document-ready	file-size	fs-promise	get-value	group-array	has-key-deep	has-value	he	identity-function	in-array	is-empty-object	is-generator	is-number	is-promise	is-sorted	left-pad	missing-deep-keys	node-du	node-slug	normalize-pkg	not-defined
SA	0.42	1.44	2.02	1.94	5.20	2.04	3.13	6.90	1.59	5.14	3.78	4.02	1.78	8.49	1.38	3.84	7.51	7.26	5.47	6.82	0.54	3.80	3.82	2.38	5.63	1.00	7.01	4.22	26.70	5.23
PD	0.23	0.42	1.93	1.41	5.61	2.13	2.19	6.31	1.87	4.75	2.24	2.60	1.05	6.94	1.79	2.75	5.78	7.63	4.90	6.01	0.71	3.41	2.91	1.12	5.71	0.62	6.36	3.24	19.86	4.88
TID	1.42	2.27	2.59	1.67	5.45	2.03	1.68	5.80	2.35	5.90	2.43	3.04	1.09	7.47	2.04	3.54	6.91	7.42	5.49	6.36	1.01	4.63	4.00	2.74	6.14	0.64	6.05	3.37	22.68	5.24

We apply Lya's `RWX` analysis to `safe-eval` (v0.3), a module intended as a sandboxed replacement to runtime code evaluation by carefully sanitizing its input prior to calling `eval`. By executing `safe-eval`'s test suite, we manually inspect the result and confirm that the code did not escape the sandbox.

CVE reports from Snyk [2], a public vulnerability database, indicate potential vulnerability and include a proof-of-concept exploit (PoC). The PoC payload breaks out of the sandbox by accessing `process` through the prototype chain, and then binding `child_proceess` to spawn a `whoami`. Using as a starting permission set the one obtained from the tests, Lya's `RWX` analysis records and reports multiple invalid accesses—*e.g.*, `R` access over the prototype chain, `X` access over `require`, and `R` permission over `child_process`.

Performance Diagnosis (87LoC): We developed a profiling analysis that operates at two levels: (i) module-boundary wrappers that collect profiling statistics for calls between modules by wrapping module interfaces; and (ii) an aggregator function that collects statistics from all boundary wrappers and generates a model of library load under the current workload. Boundary wrappers operate at a higher-frequency intervals than the aggregator, which operates on summaries. Their analysis focuses on function calls, skipping all other direct field accesses. Functions are wrapped with prologue and epilogue wrappers that record statistics from the Node.js runtime—for now, a frequency counter and a timer between prologue-to-epilogue invocations. Boundary wrappers summarize these statistics by periodically sending a windowed, weighted average of call latencies to the aggregator function.

We apply Lya's performance analysis to `uri-js` (v2.1.1), an extensible URI parsing and validation module that is fast in the average case. Issuing an HTTP load of 5Kreqs/s of `uri-js`'s test dataset, `uri-js` responds with an average latency of 34.3ms (σ: 2.8ms);

Special pathological inputs, found on public Github issues [10], can cause `uri-js` to spend upwards of 400ms per URI in pathological edge cases. Servicing a workload with 99% benign URIs and 1% pathological URIs, the throughput of the unmodified `uri-js` drops to 197req/s (15.2s per request, σ: 11.04s). Lya's analysis detects and reports the load pressure applied on `uri-js`.

Type Invariant Discovery (86LoC): To infer type invariants for serialization specifications, we based our analysis on a simple type system modeled after the simply-typed lambda calculus augmented with: (i) unions (sum types), such as `string | number`, (ii) JavaScript-specific types such as **`null`**, **`NaN`**, or **`undefined`**, and (iii) a **`native`** type for values that cannot be serialized, such as `console.log`. Support for union types is useful for when our analysis witnesses variables holding values of different types—although, in practice, programs tend to make calls of the same type across their entire lifetime [12].

We apply Lya's type invariant detection to the built-in JSON serialization module. By polling from a fixed set of different invariants, Lya can quickly detect whether a value is not likely to be processed by the module intended for that value—in particular, here it analyzes whether a provided object structure contains cycles. While objects in the test suite do not contain cycles, Lya detects multiple anomalous instances where objects contain cycles and thus require a different serialization-deserialization library.

5.2 Runtime Performance

To understand Lya's runtime performance characteristics, we performed three experiments. In the first experiment, we apply Lya on 30 popular libraries from `npm` and observe average overheads 3.6–4.1%. In the second experiment, we compare Lya with Jalangi on Jalangi's workloads, and observe 1–3 orders of magnitude speedups (average 87×).

Lya on Popular Libraries: For the first experiment, we evaluate Lya on 30 JavaScript modules from the `npm` ecosystem. These modules are from a curated list containing a list of "small, focused Node.js modules" [31], which we sort by module popularity, and pick the top 30: These modules average about 4.8M weekly downloads (total: 227M) and are depended upon by about 656 other modules or applications on average (total: 30K).

Each library was run against the test suite provided by its nominal developers, via `npm test`. As these libraries are quite popular, they have received significant investment in their testing infrastructure, resulting into two main characteristics relevant to Lya's evaluation: (i) different tests stress different parts of the library and corresponding analysis primitives; (ii) even if most applications that import these libraries use only a fraction of their functionality, tests still cover the majority of provided functionality—*e.g.*, we observed test suites that were 10× the size of the corresponding library.

Tab. 1 shows the performance overhead of applying Lya to these libraries as a percentage of the vanilla runtime. Overheads report on running each library's entire test suite under the three analyses and comparing against the non-Lya version. For the security analysis,

the average is 4.14% (max.: 26.7%; min.: 0.13%); for the performance analysis, the average is 3.62% (max.: 19.86%; min.: 0.23%); and for the type-invariant analysis, the average is 3.86% (max.: 22.68%; min.: 1.21%). To average across analyses, Lya incurs about 3.8% slowdown.

We zoom into the sources of these overheads in a later subsection (§5.3).

Lya vs. Jalangi: For the second experiment, we compare the performance of Lya to that of Jalangi. Jalangi is a popular dynamic analysis framework for JavaScript, providing fine-grained instrumentation by executing on a custom interpreter.

For this experiment, we use a different set of benchmarks—those of Jalangi itself—as we were not able to run Jalangi on the 30 micro-packages. This inability was because many of these 30 packages make use of newer EcmaScript features; examples of such features include arrow functions, template strings, destructuring, and enhanced object literals. Jalangi was further perplexed by `npm test` (which is an external program outside Node, but tightly coupled with it). Contrary to Jalangi, Lya allows the existence of recent language features, demonstrating the compatibility benefits of operating on an unmodified runtime. Jalangi's benchmark suite includes 26 programs from SunSpider [9], which we execute as part of a Jalangi-provided docker container [15]. For these experiments, Lya was also run in the same container.

In terms of analysis, both systems are configured to perform dynamic frequency analysis of accesses to global variables. The analysis is a common denominator between Lya and Jalangi, designed and implemented from scratch to ensure a meaningful comparison between the two frameworks. Such analyses are useful for understanding how program components interact with global state—*e.g.*, for generating remote-procedure stubs or scaling out functional components [38].

The performance results show that Lya performs better than Jalangi, at times by a significant margin: on average, Lya takes 87× (max.: 266.3×, min.: 3.1×) longer to complete than Lya. There are two reasons why Lya outperforms Jalangi, both of which related to Lya's main thesis. The first reason is that, to achieve its analyses, Jalangi is implemented as a custom JavaScript interpreter written in Python, which is less efficient than the native JavaScript implementation. Lya, on the other hand, operates on a completely unmodified V8 engine, Google's JavaScript runtime environment, taking advantage of the environment's just-in-time compiler, well-engineered garbage collector, and other production-grade engineering investment. The second reason is that Lya operates at a significantly coarser granularity, selectively wrapping data values needed by the analysis, whereas Jalangi instruments every sub-expression—including language-level constructs such as **`if`**, **`while`**, **`break`**, + *etc.* For these two reasons, Jalangi interprets the entire program using its own custom interpreter and hooks, whereas Lya adds only a small overhead on only the dataflows being analyzed. These differences show that language-based recontextualization transformations à la Lya can deliver non-trivial performance improvements.

5.3 Further Micro-benchmarks

This subsection presents a series of micro-benchmarks that zoom further into Lya's sources of overheads. The key results are that (i) the majority of the overhead comes from the JavaScript's **`with`** construct, only used for a small fraction of Lya's transformations—only to global variables; (ii) interposition overhead is negligible in practice; (iii) while wrapped and accessed fields increase exponentially as a function of depth (as objects have many fields), object explosion quickly plateaus around level four with under 400 fields; and (iv) the majority of wrapped and accessed fields come from Node and EcmaScript names rather than imports or global values.

Table 2: Synthetic Micro-benchmarks. Applying the three analyses on a series of synthetic micro-benchmarks, created to stress different features. All timings are in *ms* (*Cf.*§5.3).

	Base	SA	PD	TID
global vars	0.90	4.70	4.54	4.30
built-in fields	1.44	6.46	6.24	5.96
counter	3.37	6.26	6.32	5.72
all names	7.79	13.54	13.31	12.8
custom delays	24661	24848	24754	24760
direct-access	4.06	7.24	7.16	6.79
simple-types	4.11	7.25	7.23	6.86
cycles	4.32	8.32	8.19	7.73

Sources of Overhead: To understand the sources of these overheads, we perform a series of micro-benchmarks with tight loops calling several ES-internal libraries without any I/O. By enabling different parts of Lya, we discover that the primary source of overhead comes from JavaScript's **`with`** construct: disabling **`with`** makes 95% of the overheads disappear. The reason **`with`** dominates overheads is twofold: it (i) interposes on too many accesses, only a fraction of which are relevant, and (ii) remains significantly unoptimized, since its use is strongly discouraged by the JavaScript standards.

Interposition Overheads: Table 2 depicts the results of the three analyses applied to a subset of the aforementioned synthetic benchmarks. The first column indicates the focus of the benchmark; not all analysis–benchmark combinations are useful: for example, the "custom-delay" benchmark features static bottlenecks across its dependency tree but does not involve interesting access patterns. Lya-induced slowdown is under 2×, except for the first two cases that feature *only* accesses. Close inspection confirms a correlation to the number of wrapped objects and the frequency of accesses: these benchmarks feature artificially tight loops with high-frequency accesses. Transformation overheads themselves (not in Tab. 2) remain under 1ms.

To understand the costs of proxy interposition, we measure the time to access deeply-nested properties of two versions of an object: unmodified and proxy-wrapped. Paths to the properties (*e.g.*, `a.b.c....`) are random but generated prior to running the experiment. We construct 300MB-sized objects, each with a fanout of 8 fields nested for 12 levels. The proxy-wrapped version introduces interposition at every level. Traversing one million 12-edge paths (*i.e.*, root to leaves) averages 167.2ms and 595.7ms (3.5×) for the unmodified and proxy-wrapped versions, respectively. We emphasize that this is an artificially constructed benchmark stressing worst-case overheads nowhere near an normal execution: for comparison, the transformation of these objects itself took nearly 16 seconds (10^3× more than what we saw with real modules). The takeaway is that Lya-inherent overheads due to interposition are unlikely to

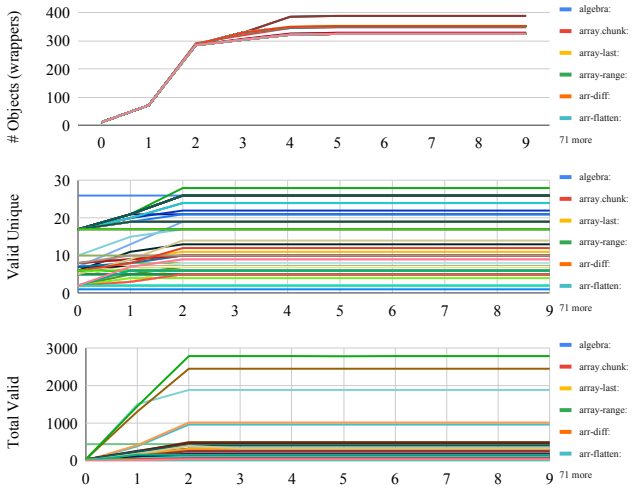

Figure 3: Analysis characteristics as a function of the analysis depth. From the top: (i) number of Lya wrappers applied, (ii) number of unique accesses (*i.e.*, counting each access once), (iii) total number of accesses (*Cf.*§5.3).

be the bottleneck of an analysis; what is likely to be is the analysis itself—*e.g.*, updating a global aggregator or invoking system calls to extract timing.

Analysis Depth: To understand the effect of depth in practice, Figure 3 presents the number of wrapped object, unique accesses, and total accesses as a function of depth for all 30 libraries (§5.2). Depth is the distance from the root of each name path up to the last accessible field and represents how deep Lya traverses references starting starting from the names in scope—*e.g.*, the access `global.obj.x` is two levels deep and `fs.readFile` is one level deep.

There are a few highlights worth noting. While the number of objects wrapped by Lya starts growing rapidly, it quickly reaches an average upper bound of 400 (depending on the exact benchmark). Accesses grow exponentially for the first couple of levels—as objects at levels have multiple fields, many of which are accessed several thousand times—but then stabilize around level 5. This is because most interfaces follow a mostly-flat format where all methods are defined at the top level or right under.

Context: Context refers to the broad source of names that are available in the current scope—ones defined by the EcmaScript standard (es), through an explicit import (exports), by the Node.js runtime (node), or via global variables (globs). A few names seem globally available but are in fact module locals (`require`). User-defined global variables are not prefixed with `global` thus requiring special interposition (**`with`**).

Tab. 3 shows Lya's context characteristics on all 30 libraries (§5.2). In terms of the number of objects wrapped, the majority comes from Node (91.1% of all wrappers). In terms of unique number of accesses, for both invalid and invalid the majority comes from `require`; taking their number into account, valid accesses concentrate on ES and Node, whereas invalid ones concentrate on **`export`**s.

Table 3: Access characteristics as a function of context. Rows: (1) number of Lya wrappers, (2) unique valid accesses (*i.e.*, counting each access once), (3) total valid accesses (*Cf.*§5.3).

	es	require	exports	node	globs	with
Object Wrappers	215	610	853	24260	531	158
Unique Valid Accesses:	52	611	108	273	54	0
Total Valid Accesses:	52	1556	110	13982	52	0

6 DISCUSSION, LIMITATIONS, & THREATS TO VALIDITY

This sections discusses several aspects related to the design, implementation, and evaluation of module recontextualization.

Runtime Environment Modifications: A key benefit of the analysis approach presented in this paper is that there is no need to modify the runtime environment. This leads to important performance and compatibility benefits discussed in the evaluation section, but in principle can also lead to significant usability benefits: developers do not need to setup and use a modified version of the runtime system different and possibly divergent from the version of the runtime system they normally use.

Security Implications: On the surface, the changes to the module system,*e.g.*, in JavaScript or Racket, of the analysis approach presented in this paper might seem as affecting the security properties of the overall framework—including the inference and enforcement of specific security policies expressed in Lya. In principle, however, these security implications are no different from the ones of applying these changes in the underlying runtime environment itself. On the contrary, applying these changes to the lower-level, memory-unsafe, and type-unsafe language of the runtime environment implementation itself—*i.e.*, C/C++ for V8—would result in a higher risk of insecurity. We also note that different security-related analyses and instrumentations are developed in response to different threat models; thus, understanding whether a modification—irrespective of the level applied, *i.e.*, that of the module system or the runtime environment—is secure with respect to a particular threat model would not be conclusive without considering the specific analysis or policy at hand.

Other analyses: Module recontextualization is well-suited for analyses on field-granularity read/write access and/or function-granularity control flows, and especially ones that might be needed in customer-facing production environments (not testing) such as runtime subversion, denial-of-service detection, and coarse-grained taint tracking. It is not well-suited for analyses that operate at the granularity of language-level constructs such as **`if`**, **`while`**, **`break`**, + *etc.*—but could be used even then to narrow down the search before applying more heavyweight analyses. We note that Lya's analysis hooks allow for Turing-complete code, including access to state not visible to the code being analyzed but shared among all hooks comprising an analysis. This allows analysis code to implement powerful security or performance monitors beyond the ones presented in the current paper.

Semantic preservation: Analyses that focus on measurement preserve the semantics of the original application, because Lya's

transformations do not interfere with the runtime execution of the transformed values: wrappers simply forward calls to internal functions, observing but not altering the call arguments. However, when the intention of the instrumentation is to alter the behavior of the program, wrappers interpose to introduce corrective behaviors not present in the original program: a typical example is the enforcement of security policies *e.g.*, monitoring access to sensitive values and intervening to block unauthorized accesses.

Language-specific hooks: One limitation of language-specific module recontextualization is that it requires developers to write the same analysis in as many languages as the applications they want to analyze. We note that conventional analysis frameworks, such as Jalangi, may have this property too; each of these frameworks—Lya included—could expose a DSL for writing analysis-specific code in a language-agnostic fashion. However, language-specific hooks have several significant benefits: they (1) preserve developer knowledge, expertise, libraries, and code; and (2) leverage the semantic correspondence between the code implementing the analysis and code being analyzed—for example, cooperative *vs.* preemptive concurrency, prototype-based *vs.* class-based inheritance *etc.*

Top-level Scripts: A limitation of the Lya implementation is its inability to analyze the library importing Lya—usually, the top-level program entry point equivalent to `main`. This is because Lya cannot transform the context of the top-level script, because the context has already been loaded and bound to the interpreted code (and which has also been interpreted). As a result, Lya as-is cannot be applied to analyze single-file programs—a pattern that is not unusual in scripting languages, often used for quick-and-dirty tasks. The simplest workaround we have found is to create an auxiliary file that (1) imports Lya, and then (2) imports (and invokes, if that is not achieved by the import) the single-file script.

Outperforming Lya: The likelihood of fine-grained analysis frameworks such as Jalangi outperforming coarse-grained analysis frameworks such as Lya, especially on more complex analyses, is a possible threat to the validity of the results. Based on (i) our understanding of the techniques involved, and (ii) the data accumulated through extensive use of Lya and Jalangi, we do not foresee a situation in which this would occur—especially in more complex analyses. Jalangi sits at a different design point than Lya: it operates on a custom Python runtime and at a very fine granularity—both of which result in order-of-magnitude differences in overhead on real analyses.

7 RELATED WORK

Aspect-oriented programming (AOP) is a programming model in which program points (or more generally queries against the program trace) map to actions taken at these points [17]. Aspects are typically implemented via explicit language extensions (e.g. AspectJ and AspectC++) and/or via modifications to the original language implementation or runtime system. Lya, in contrast, leverages the existing dynamic loading and metaprogramming capabilities in modern dynamic languages to operate completely within unmodified production language runtimes.

There are several dynamic analysis frameworks for JavaScript [5, 16, 33, 36]. These systems allow much more fine-grained analyses, including tracking language-level constructs such as `if`, `while`, `break`, + *etc.* Their goal is thus different from Lya's, which focuses on coarser but online analysis and enforcement.

NodeProf [36] is a fine-grained dynamic analysis that uses AST instrumentation to insert analyses. While it supports finer-grained analysis than Lya, it works with the underlying Graal [41] and Truffle [42] APIs. Graal is compliant with, but different from, Node.js, and thus NodeProf does not target unmodified Node.js. runtimes.

Dynamic instrumentation frameworks [8, 13, 22, 24, 26] wrap basic blocks of a program incrementally and right before execution, similar in vein to how Lya wraps libraries. They operate at a much lower level (binary) than Lya, are much more detailed and heavyweight, and are usually not available to high-level languages as a language-aware library.

JavaScript is related to WebAssembly, a standardized subset of JavaScript target designed to serve as a compilation target. The first dynamic analysis framework for WebAssembly, Wasabi [19], shares some of Lya' goals—*e.g.*, low-effort analysis and API for observation rather than manipulation. Contrary to Lya, Wasabi instruments binaries statically, *i.e.*, ahead-of-time, and aims for heavier-weight higher-resolution instrumentation.

Lya draws inspiration from data-oriented JavaScript analysis and query systems for the web [25, 27, 28]. Contrary to them, Lya applies source-to-source transformations to add custom modified contexts for production analysis and instrumentation.

Lya is related to program fracture and recombination (PFR) [1, 34], a line of work less tied to program analysis and more towards program synthesis and automated patch generation. PFR breaks up multiple programs into many components with the goal of exchanging functionality between donor-donee pairs of programs. Contrary to PFR, Lya operates on single programs, avoids breaking semantics, and leverages the existence of components with (mostly) explicit boundaries in the guise of modules.

8 CONCLUSION

This paper presented module recontextualization, an efficient module-level dynamic analysis technique, and Lya, an implementation for JavaScript and Racket. Lya decomposes, transforms, and reassembles programs by combining techniques for name shadowing, context re-binding, and load-time transformation of the underlying dependency graph. It delivers order-of-magnitude performance improvements over state-of-the-art dynamic analysis systems while supporting a range of useful analyses, each implemented in about 100 lines of code. Lya is available—for installation and experimentation with other applications and analyses—as open source:

https://github.com/andromeda/lya

ACKNOWLEDGMENTS

We thank Sage Gerard, Sotiris Ioannidis, Konstantinos Kallas, Ben Karel, Michail G. Lagoudakis, Mary McDavitt, Jeff Perkins, and MIT CSAIL's PAC group. The term "recontextualization" was used by Ania Vu in a musical context, then recontextualized by the first author. Much of Lya's design was informed by interactions with the broader community. We are particularly thankful to Isaac Z. Schlueter and CJ Silverio from npm and Petros Efstathopoulos, Daniel Katz, Daniel Marino, and Kevin Roundy from Symantec/NortonLifeLock Research Group. This work was partly supported by DARPA contract no. HR00112020013, HR001120C0191, and HR001120C0155.

REFERENCES

[1] Peter Amidon, Eli Davis, Stelios Sidiroglou-Douskos, and Martin Rinard. 2015. Program fracture and recombination for efficient automatic code reuse. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[2] Anirudh Anand. 2020. *Sandbox Escape: Safe Eval.* https://snyk.io/vuln/SNYK-JS-SAFEEVAL-608076

[3] 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).

[4] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. 2019. Active Learning for Software Engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 62–78. https://doi.org/10.1145/3359591.3359732

[5] Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. 2015. Poster: Dynamic Analysis Using JavaScript Proxies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 813–814. http://dl.acm.org/citation.cfm?id=2819009.2819180

[6] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, USA.

[7] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.Js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 343–359.

[8] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. 2019. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 693–708. https://www.usenix.org/conference/atc19/presentation/demoulin

[9] Webkit Developers. 2020. *SunSpider 1.0.2 JavaScript Benchmark.* https://webkit.org/perf/sunspider/sunspider.html

[10] Peter Dotchev. 2016. *Parse hangs on some long urls.* https://github.com/garycourt/uri-js/issues/12

[11] Ayrton Sparling *et al.* 2018. *Event-Stream, GitHub Issue 116: I don't know what to say.* https://github.com/dominictarr/event-stream/issues/116

[12] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1–3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[13] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[14] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. Unix Shell Programming: The Next 50 Years. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 104–111. https://doi.org/10.1145/3458336.3465294

[15] Hrishikesh. 2018. *Dockerhub: Jalangi Docker Container.* https://hub.docker.com/r/hrishikeshrt/jalangi

[16] Matthias Keil and Peter Thiemann. 2013. Efficient Dynamic Access Analysis Using JavaScript Proxies. In *Proceedings of the 9th Symposium on Dynamic Languages* (Indianapolis, Indiana, USA) *(DLS '13)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2508168.2508176

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 220–242.

[18] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. (2017).

[19] 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

[20] SS Jeremy Long. 2015. OWASP Dependency Check. (2015).

[21] Snyk Ltd. 2018. *minimatch@2.0.10.* https://snyk.io/test/npm/minimatch/2.0.10

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[23] Michael Maass. 2016. *A Theory and Tools for Applying Sandboxes Effectively.* Ph.D. Dissertation. Carnegie Mellon University.

[24] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-Specific Language for Bytecode Instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development (AOSD '12)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/2162049.2162077

[25] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 11.

[26] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[27] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster page loads using fine-grained dependency tracking. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*.

[28] Ravi Netravali and James Mickens. 2019. Reverb: Speculative Debugging for Web Applications. In *Proceedings of the ACM Symposium on Cloud Computing*.

[29] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated.

[30] npm, Inc. 2018. *Details about the event-stream incident.* https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident

[31] Andrea Parodi. 2020. *Awesome Micro npm Packages.* https://github.com/parro-it/awesome-micro-npm-packages

[32] Niels Provos. 2003. Improving Host Security with System Call Policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, USA, 18.

[33] 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* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[34] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer across Multiple Applications. *SIGPLAN Not.* 50, 6 (June 2015), 43–54. https://doi.org/10.1145/2813885.2737988

[35] Snyk. 2016. *Find, fix and monitor for known vulnerabilities in Node.js and Ruby packages.* https://snyk.io/

[36] 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* (Vienna, Austria) *(CC 2018)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/3178372.3179527

[37] The gRPC Authors. 2018. *gRPC.* https://grpc.io/ Accessed: 2019-04-16.

[38] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. 2019. Ignis: Scaling Distribution-Oblivious Systems with Light-Touch Distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1010–1026. https://doi.org/10.1145/3314221.3314586

[39] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security* (San Diego, California) *(NDSS'18)*. https://doi.org/10.14722/ndss.2018.23131

[40] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Seoul, South Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA.

[41] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöundefined, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. *SIGPLAN Not.* 52, 6 (June 2017), 662–676. https://doi.org/10.1145/3140587.3062381

[42] Thomas Würthinger, Christian Wimmer, Andreas Wöundefined, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[43] Serdar Yegulalp. 2016. *How one yanked JavaScript package wreaked havoc.* http://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html

[44] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with High Risks: A Study of Security Threats in the Npm Ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 995–1010.