LYA: Library-Oriented Program Analysis

Nikos Vasilakis Grigorios Ntousakis* Martin Rinard
MIT, CSAIL **TU Crete, ECE*

Abstract

Dynamic program analysis is a long-standing technique for obtaining information about program execution. We present a new dynamic analysis approach that targets modern dynamic languages such as Python, Lua, and JavaScript, 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 library in unmodified production language runtime environments. We implement this approach in LYA, a system targeting the JavaScript ecosystem. Results show that LYA delivers over an order of magnitude performance improvements over current state-of-the-art dynamic analysis systems, compatibility with over 50 libraries and applications, and support for a range of dynamic analyses, each implemented in under 100 lines of code.

1 Introduction

Dynamic analysis is a type of analysis performed by (and while) executing a program, with the goal of extracting information about the program and its execution. Such information may include the ability to infer execution invariants, check security constraints, and extract performance characteristics [33]. Contrary to other types of analysis, its key benefits make it the primary (or only) candidate in a variety of scenarios—namely, ones that require: (i) current runtime information, such as profiling data, in view of changing load patterns [10], (ii) accurate visibility into the execution, devoid of imprecision or approximation [10], (iii) Turing-complete policies, where monitors themselves are programs [8, 29, 31], (iv) dynamic or runtime-reflection features, such as the ones present in Python and JavaScript [37, 48].

Unfortunately, reaping these benefits comes with significant costs in terms of developer effort or runtime performance. Manually instrumenting a program requires a good understanding of its internals. External tools such as instrumentation frameworks [26, 32] and performance tracing tools [6, 14] only add to the curve, as they feature their own language for specifying analyses. Modifying the application's runtime environment such as the interpreter is cumbersome, error-prone, and requires development in a language that is different than that of the program. Tools that virtualize execution [13, 26, 32, 40] have high performance costs—for example, Jalangi [40] and RoadRunner [13] report No-Op analysis on the order of 26–32$\times$ and 52$\times$, respectively.

Effort and performance costs are severely compounded by the use of third-party libraries,[1] often glued together without full understanding of their internals [3]. Applications frequently count hundreds of libraries [53], each one often only a few lines long [52, 53]. Library-oriented analysis adds to the challenges of prior tools as (i) library boundaries disappear at runtime, and (ii) coarse-grained, high-level language semantics misalign with fine-grained, low-level instrumentation.

This paper presents a novel approach to dynamic analysis, targeting dynamic programming languages—and implemented in LYA for the server-side JavaScript ecosystem (>1.3M libraries [7]). LYA leverages the ability of these languages to import code at runtime as a string in order to apply transformations that wrap the module about to be imported with developer-provided, analysis-specific code. More specifically, LYA inserts a series of steps in the module-loading process result in wrapping a library's full observable behavior—rather than just its interface—and exposing it to developers for runtime analysis. LYA effectively fragments, transforms, and reassembles the application at the level of individual libraries. The overall approach has several practical benefits over prior work: (i) it allows bolting the dynamic analysis as a library onto unmodified runtime environments, (ii) it supports analyses expressed using the same language, tools, and abstractions as the program language, (iii) it features low runtime performance overheads, enabling toggling its use in

[1]The terms library, package, and module are used interchangeably.

production environments.

To demonstrate that LYA can be useful in practice, despite its coarse granularity, we develop a diverse set of analyses: a read-write-execute security analysis, a profiling analysis for identifying bottlenecked modules, and an analysis extracting union-based type invariants. Each of these analyses takes about 100 LoC in JavaScript; this integration between the code being analyzed and the one implementing the analysis has significant usability implications. Finally, LYA's performance is evaluated extensively, using (i) over 50 popular libraries and scripts, highlighting compatibility and performance improvements over state-of-the-art frameworks; (ii) three full applications, demonstrating that LYA scales to large and complex programs; and (iii) a series of micro-benchmarks that zoom into a few specific characteristics of LYA. §2–5 present our key contributions:

- §2 characterizes the shared needs of three case-study multi-module analyses whose needs remain mostly by general-purpose analysis frameworks.
- §3 presents the design of LYA, bolt-on library-oriented dynamic analysis, which meets the requirements of the three case-study analyses.
- §4 discusses our implementation, LYA, as a pluggable library for the JavaScript ecosystem, as well as the implementation of the three analyses.
- §5 evaluates LYA, showing that it expresses insightful analyses succinctly, adds minimal overheads over the baseline runtime, and scales to hundred-library programs.

The paper then discusses LYA's limitations and its application to other languages (§6); it compares with related prior work (§7); and draws appropriate conclusions (§8).

2 Background, Examples, and Goals

A single application today often incorporates multiple libraries written and published by several different authors. To make the difficulties of analyzing such library-overreliant software more concrete, we describe three example analyses (§2.2). These examples illustrate key requirements for the design of a framework for analyzing applications at the boundaries of modules (§2.3). Before describing these examples, however, we offer a brief refresher on the internals of a module system (§2.1).

2.1 Background on Module Systems

Libraries are development-time construct encapsulating reusable functionality. This functionality 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. Fig. 1 shows a

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

print(r);

exports = r;
```

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

Fig. 1: Example of a library import. The program on the left imports the `simple-math` library, shown right, which imports `log` and exports a `Math` object. Several names are provided by the programming language: `import` and `exports` resolve to library-local values; `print` resolves to a global value.

`simple-math` library (right) imported by a program (left).

From a developer's perspective, `import`ing a library 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 library 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 library may `import` other libraries, cause side-effects to the file-system or the network, or even be implemented in multiple languages.

From the programming language's perspective, the process of `import`ing a library is achieved in several steps. Compiled languages support separate compilation, where source files are compiled individually, followed by a runtime-linking phase that loads and links the resulting object files. Interpreted languages involve a somewhat convoluted process that combines runtime loading, interpretation, and linking. This process starts by locating the library in the file system, by resolving code-relative module identifiers. The content of the library is then read and wrapped in a way that resolves library-local names, such as `__name__` in Python and `__filename` in JavaScript, 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 library'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.

There is significant variation, but a few features are common across many implementations. A library cache maintains a mapping from library identifiers, often absolute paths, to returned `export` values. The cache serves a dual purpose: by avoiding locating already loaded libraries, it improves performance and ensures the consistency of any library-internal state. Recursive imports are handled in a depth-first way, and when cyclical dependencies are possible, they are resolved by pointing to the cache. An increasingly common feature is to allow different versions of the same library to co-exist in a program, in order to avoid a choice between mutually exclusive options—a paradoxical situation known as "dependency hell". As a result, a single `import` l does not necessarily resolve to the same (version of the) library l every time. The dual of this is also possible: two different library names may resolve to the same identifier (*i.e.*, point to the same cache entry). These features complicate library-level analysis, especially in interpreted programming languages.

2.2 Motivating Examples

Having reviewed the building blocks and underlying techniques that power modularity, we now turn to examples of dynamic analysis that today are difficult or require specialized solutions: (i) a read-write-execute security analysis, (ii) a performance-profiling analysis, and (iii) an analysis extracting runtime type invariants. These examples illustrate key design requirements for LYA.

Security Analysis The pervasive reliance on third-party libraries has led to an explosion of supply-chain attacks [21, 24, 27, 42]. Both bugs and malicious code in libraries create attack vectors, exploitable long after libraries reach their end-users. As library boundaries disappear at runtime, libraries end up executing with no meaningful isolation or privilege separation guarantees between each other and the trusted portions of a program. Popular libraries, depended upon by tens of thousands of other libraries or applications, allow vulnerabilities deep in the dependency graph to affect a great number of applications [52, 53]. Discovered vulnerabilities are becoming harder to eradicate, as updates are fetched automatically [38] and module unpublishing is becoming a multi-step process to avoid breaking applications [49]. Worst of all, leaked publishing tokens allow attackers to update packages with code that will eventually reach end-users via package updates [34].

As a concrete example, consider the recent `event-stream` incident [35, 43], in which a maintainer of a highly popular library inserted code stealing Bitcoin wallet credentials from programs using that library. Heavy-weight runtime instrumentation [40] would not have helped, as `event-stream` activated only on production (rather than testing or development). Coarse-grained analysis at program boundaries, say via containment or system-call interposition [36], 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.

Having the ability to toggle a library-level "allow-deny" security analysis on/off at runtime would have revealed the (unusual) resources touched 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. Libraries are written by authors with different education, backgrounds, quality standards, and, most importantly, needs: it is not unusual for a weekend project solving a local need to end up as a critical dependency among several open-source projects [9]. Maintenance and rewriting can also cause performance regressions between two versions of a library; worse even, the fact that a library changes one of its dependencies can cause severe performance pathologies to upstream projects—that is, while an interface remains the same due to an intermediate layer of abstraction, its performance can change dramatically [15].

As a concrete example, consider the `minimatch` library, a regular-expression-based file-matching utility susceptible to long delays due to regular expressions that involve backtracking [25]. Pathological inputs reaching `minimatch`, even if benign, can cause significant performance degradation [4] deep in the dependency chain, affecting also other parts of the program competing for the same resources [5]. Developers use various techniques to understand such problems—*e.g.*, collecting and replaying traces against off-line 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.

Switching-on a library-level profiling analysis would quickly detect a slowdown and 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 in at the library boundary is helpful in a variety of scenarios. Types can be used to (i) identify library invariants to be preserved during code modifications, (ii) generate (machine-checkable) library documentation, or (iii) guide library synthesis and regeneration [11]. This is especially important in dynamic programming languages: in the absence of static type signatures, there is not much one can say about a library without executing it and inspecting its properties.

As a case in point, consider the `gRPC` library for serializing and de-serializing objects [45]. To use it, developers have to 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.

Library-level dynamic analysis can be used to discover such type assertions or invariants. The analysis would consult the (static) 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 Design Goals

We have briefly introduced three motivating analyses for extracting information about an application's dependencies. While the challenges behind these analyses share several characteristics, today they remain largely unaddressed by dynamic analysis frameworks due to a combination of factors: these frameworks either require highly specialized solutions often outside the core language, do not leverage nor offer semantic information at the level of library boundaries, or lead to a high (often impractical) performance overheads due to the high granularity of their analyses. To summarize, these applications would appear to be served well by a system that:

- **(G1)** Operates at the level of libraries, leveraging both the semantics and performance potential that this level of granularity can potentially offer.
- **(G2)** Does not require learning a new tool or language, but rather allows developers to express their analyses in the same language as the program.
- **(G3)** Can be implemented as a library that bolts analyses onto an existing language runtime environment.

The next section describes the design of LYA, a new dynamic analysis framework that satisfies these goals, in a language-agnostic manner (§3); the section following that describes LYA's concrete implementation for server-side JavaScript (§4).

3 The Design of LYA

LYA starts by dynamically modifying the functionality of the module system that is responsible for importing and loading libraries: instead of simply locating and loading a library from the file-system, it yields control to LYA, which applies a series of transformations to libraries 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 dismantling the program at the boundaries of libraries, applying transformations that insert analysis-specific machinery, and then carefully reassembling individual components to maintain the original semantics:

- **Fracture:** LYA starts by recursively dismantling a program into its dependencies. This is achieved by re-wiring 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 libraries, a specific library *etc.*) in order to apply transformations at the correct level and map the provided analysis hooks to the corresponding libraries.
- **Transformation:** It then sets up the provided analysis, by transforming each library interface, its surrounding environment, and possibly the values passing through the library 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 library, and swapping all original values externally-available to a library with ones that are wrapped with interposition mechanisms.
- **Reassembly** Finally, LYA reassembles individual modified libraries back into the program's original structure. The most important challenge during this phase is the treatment of the library cache (§2.1). The cache needs to be augmented to support multiple wrappers per library, each capturing a part of the overall analysis.

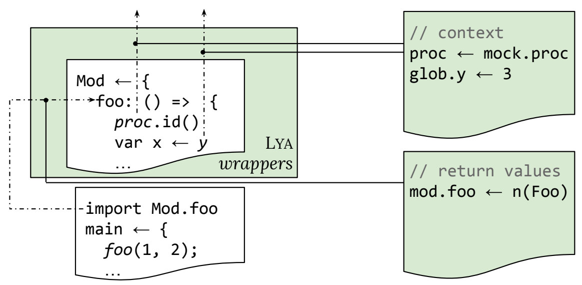
```
wrap (e: Value, α: Analysis) : Value := match e with
  | {(s, v) :: vs}  →  {(s, wrap v) :: wrap vs}
  | [v :: vs]       →  [(wrap v) :: wrap vs]
  | λ(...args).f    →  λ(...args).{ α₁( f( α₂(args) ) ) }
  | _               →  interpose(α₃, e)
end
```

Fig. 2: Base transformation. 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.

3.2 Fracture

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 it 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



```
Mod ← {
  foo: () => {
    proc.id()
    var x ← y
    ...
```

```
import Mod.foo
main ← {
  foo(1, 2);
  ...
```

```
// context
proc ← mock.proc
glob.y ← 3
```

```
// return values
mod.foo ← n(Foo)
```

Fig. 3: Shadowing segments. Cross-module variable name resolution (left) augmented with LYA (green boxes), which interjects non-bypassable steps resolving to LYA-augmented values (right-top: implicit module imports; right-bottom: explicit import) (*Cf.*§3.3).

`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.

3.3 Transformation

For each analyzed library, LYA needs to place hooks all around its boundary—not just its interface (Fig. 3). 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, however, 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 (Fig. 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 and §6.1 for equivalents in other environments). 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.

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[2] 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

[2]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`.

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
...
```

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 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.[3] 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 the library 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 Implementation: Server-Side JavaScript

This section details the implementation of LYA for server-side JavaScript (Node.js v8.9.4) in about 1.5K LoC (§4.1), and the three analyses developed atop LYA (§4.2). The differences between JavaScript and other languages are left for §6.1.

The aforementioned LYA design can be implemented in either of two ways. The first is to implement LYA 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, LYA feels to users as a backward-compatible, drop-in replacement of Node's module 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 Node copy only for running analyses; it removes LYA's developers from the critical path of updates between the Node developers and its users; and it simplifies LYA's comparison with vanilla Node (both in terms of performance and correctness). The primary drawback was missing a few opportunities for lowering runtime-performance and development-effort.

4.1 System Implementation

Node's module system is implemented entirely in JavaScript, exposing a library-local `require` function for importing modules. Loading a fresh module corresponds to the following

[3]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.

```
Math = {
…
mul: (a, b) => a * b,
div: (a, b) => {
 import(“info”).info(b);
 return a / b,
} };
```

(a) a `Math` object

```
let _ = Math, Math = {};
  …
Math.div = (…args) => {
 let p = lya.prologue(args);
 let v = _.div(args);
 return lya.epilogue(p, v);
}
…
```

(b) Object transformation

```
var ctx = {
 print:  lya.print,
 import: lya.txf(
            import,
            lya.a.prologue,
            lya.a.epilogue),
 (…150 ES6 entries…)
}
```

(c) Custom context creation

```
function (cxt) {
 var print = cxt.print;
 var import = ctx.import;
  (…original Math code…)
  div: (a, b) => {
    log.info.(b);
    return a / b,
  }
}
```

(d) Context rebinding

Fig. 4: Example transformations. Applying runtime transformations (b) and context rebinding (c, d) on a simple `Math` object (*Cf.*§4).

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 Node's 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 purposes of consistency and performance.

LYA augments all of these steps. Interposing on resolution (R) makes the module identifier available to LYA without affecting the (incredibly convoluted!) 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 replaces Node's wrapper function to pass an additional argument, the modified context `CTX`. LYA 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.

We found it useful to add an option for white- and blacklisting 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, which in Node are represented as 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 blacklisting built-in libraries or white-listing only the library imported most recently.

Fig. 4 exemplifies LYA's transformation techniques in the context of JavaScript. LYA traverses a `Math` object, replaces functions such as `div` with wrappers, creates a modified version of the surrounding context, and binds it to the library.

4.2 Individual Analyses

Individual analysis are implemented using a set of templates that provide initial constructs. LYA offers a small utility library to reduce boilerplate code for pattern-matching on types. Individual analyses are about 60 lines of code, but a significant part of this code is common across them.

As the three analyses are too complex to include in the paper, we present here a trivial analysis—one that counts all accesses to global variables from a module `serial`:

```
let fs = require("fs");
let count = {};
forevery.global.in(/serial/).do({
  pre: (name, path, _) => {
    let o = resolve(name, path);
    count[o] = count[o]?  count[o] + 1 : 1;
  } });
process.on("exit", () => {
  fs.writeFileSync("acl.json",
                   "utf-8",
                   JSON.stringify(count)); });
```

LYA-provided `forevery` generates sets of names succinctly. The `in` 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.

Security Analysis To address the security concerns of third-party libraries, we developed an on-off 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-like model where individual fields are governed by permission sets containing combinations of on and off permissions. At the start of the analysis all permissions are set to off (*i.e.*, default-deny), and gradually turn on based on the accesses they detect. 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: ON`.

Performance Diagnosis 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.

Interestingly, profiling functions at the boundary cannot "see through" libraries to correctly attribute load among a library and its dependencies. For example, a boundary wrapper `A` of a dependency tree `A`$\rightarrow$`B`$\rightarrow$`C` does not know how much of the latency comes from `B`. Fortunately, the aggregator understands the dependency structure and can approximate how much of the latency comes from each module.

Type Invariant Discovery 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 [11].

Our implementation also uses a base set of JavaScript types and rules to deconstruct complex types into their elementary building blocks. For example, it unpacks a type `user` into two `string`s (first and last name) and a `number` (age).

5 Evaluation

At a high level, we are interested in three main questions:

- **Q1** Given its coarse granularity, is LYA useful? We answer this question in §2, which shows three case-study multi-module analyses with practical implications. LYA operates at the right level of abstraction for answering important questions raised by today's software engineering practices.
- **Q2** Is it usable? We answer this question in §4, by implementing these and other analyses. LYA operates as a library around unmodified runtime environments, does not require investment in a new language, and leads to analysis expressions that are comparable to conventional frameworks.
- **Q3** How does it perform? We answer this question here. For the most libraries, LYA overheads are under 5% (§5.1). LYA is over an order-of-magnitude faster than Jalangi, a state-of-the-art dynamic analysis framework (§5.1). By executing on unmodified runtimes, LYA has significantly better compatibility: to execute, Jalangi requires significant library modifications (§5.1). LYA scales to three large applications with hundreds of libraries while interposing on several thousands of objects (§5.2). Most of the overheads ($> 95\%$)come from tracking global variables, but loading-time wrapping and runtime interposition overheads are negligible (§5.3)

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 v2, Node.js v8.9.4 (bundled with V8 v6.1.534.50, LibUV v1.15.0, and npm v6.4.1), 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 Single-Library Benchmarks

To understand LYA's performance characteristics at the level of individual modules, we performed two experiments.

LYA on Popular Libraries For the first experiment, we evaluate LYA on 50 JavaScript packages from the npm ecosystem. These libraries are highly popular, averaging 4.8M weekly downloads (total: 227M) and are depended upon by about 656 other packages or applications on average (total: 30K). These libraries are relatively small—adhering to known ecosystem trends [53]—simplifying compatibility inspection.

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 benefits for 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$\times$ the size of the corresponding library.

Fig. 5 shows the performance overhead of applying LYA on these libraries. Overheads report on running each library's entire test suite under the three analyses and comparing against the non-LYA version. On average, LYA's analyses add about 3.3ms per library causing 1.93% slowdown; relative overheads between analyses are negligible (avg: <0.1ms). For the largest of these libraries, we perform a lighter global-access analysis (not shown in Fig. 5). This is slightly faster, adding a 3.2ms causing a 1.62% slowdown. We zoom into the sources of these overheads later (§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. We note our use of Jalangi's v2.2.5 version: the v2 branch

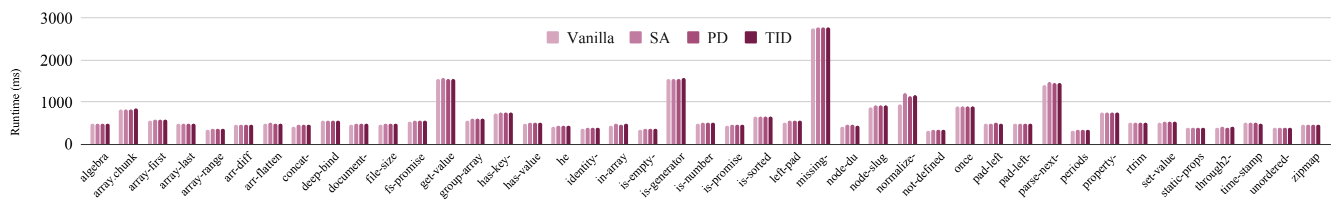


Fig. 5: Runtime overhead. The plot shows the runtime overhead of the three analyses for 50 libraries against normal (vanilla) execution (*Cf.*§5.1).

improves significantly over the performance of the original framework [40], as it eschews record/replay capabilities.

In terms of programs, we use Jalangi's extensive test suite within a Jalangi-provided docker container [17] (For these experiments, LYA is also run in the same container.) This is not entirely by choice: Jalangi's interaction with the bare-bones libraries was hitting EcmaScript 6 parsing challenges, further complicated by `npm test` (which is an external program outside Node, but tightly coupled with it). Contrary to Jalangi, LYA executes all programs, demonstrating the compatibility benefits of operating on an unmodified runtime.

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 program components interact with global state—*e.g.*, for generating remote-procedure stubs or scaling out functional components [47].

Fig. 6 presents the comparison between LYA and Jalangi. The core of the analysis requires a few lines, but expressing it in LYA requires 4× fewer lines of code. The performance results show that LYA performs better than Jalangi, at times by a significant margin (*i.e.*, more than 10× for 13 benchmarks).

Implemented as a custom interpreter, Jalangi exists at a different point in the trade-off space than LYA. On the one hand, it is significantly more fine-grained than LYA, capable of expressing and executing analyses at the granularity of individual expressions. On the other hand, it results in significantly higher incompatibility and runtime overhead—even when the analysis does not make use of its features.

5.2 Application Benchmarks

We apply LYA on three large applications: KoaBlog [16], a minimal blogging platform built on the Koa framework; Moeda [46], a command-line currency converter that, among other features, uses online currency exchange rates; and Terminalizer [12], an application for recording terminal sessions to generate animated gif images and shareable web-player links. These applications are highly popular, with Koa and Terminalizer having 28.3K and 8.7K GitHub stars respectively. Their structural and performance characteristics are shown in Tab. 1.

The differences between the number of sub-modules detected statically (by `npm ls`) and the ones observed by LYA are due to a few reasons. One is that LYA does not report accesses if a library is not accessed at runtime, whereas `npm` traverses and reports on the entire dependency structure. LYA analyzes modules that are part of the standard library (some of which have themselves internal dependencies), not shown by `npm`. Finally, LYA generates several different return wrapped objects for the same libraries (§3.4).

Applying LYA to these applications generates analysis results that are 9–132× larger than the ones for single libraries. However, the performance of analyzed code is on average still within 2–3× over the baseline performance. The size of the resulting JSON files raises the challenge of analysis interpretability—just by running Terminalizer with `-help` returns more than 6K fragments from several (hundreds of) libraries. Custom `onExit` analysis routines post-process LYA's results to compress by module name—*i.e.*, squashing duplicate reports by human-readable name, not file-system module identifier. This can affect the results of multiple imports with the same name at different parts of the tree that would indeed correspond to different libraries. We also squash cache entries that correspond to one module imported by multiple modules, which would create separate entries in LYA's augmented cache, but results in a single count for Tab. 5.2 (for LYA's treatment of the module cache, see §3.4). Confirming that LYA performs correctly under these scenarios is one part of the motivation behind the synthetic micro-benchmarks shown earlier (§5.3).

Tab. 1: Applying LYA to three large applications. The table shows the characteristics and runtime overheads of applying the three analyses to three large applications (*Cf.*§5.2).

	KoaBlog [16]	Moeda [46]	Term/zr [12]
Structure			
Size (MB)	41	29	262
Code Size (KLoC)	127	79	39
Top-level Modules, statically	24	7	33
Total Modules, statically (npm)	432	260	1084
Total Modules, dynamically (LYA)	1507	729	2355
Performance (s)			
Baseline	0.97	1.27	0.754
Security Analysis	1.48	2.00	3.346
Performance Diagnosis	1.30	2.684	2.684
Type Invariant Discovery	1.32	2.036	2.615

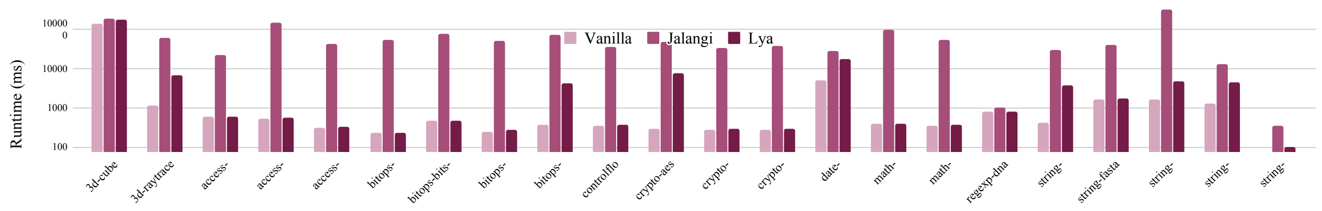


Fig. 6: Lya vs. Jalangi. The plot compares single-library programs taken from Jalangi between three setups: (i) Vanilla JavaScript, (ii) global-use analysis with Jalangi, (iii) global-use analysis with Lya. Global-use analysis is a simple analysis that tracks accesses of global variables in the program (*Cf.*§5.1).

Tab. 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

5.3 Micro-benchmarks

Lya's micro-benchmarks zoom into Lya's sources of overheads. The key results are that (i) the majority of the overhead comes from the JavaScript's `with` expression, which Lya uses to wrap the context of libraries; (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.

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 500MB-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 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, Fig. 8 presents the number of wrapped object, unique accesses, and total accesses as a function of depth for all 50 libraries (§5.1). Depth is explained in the previous paragraph: it is about 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`).

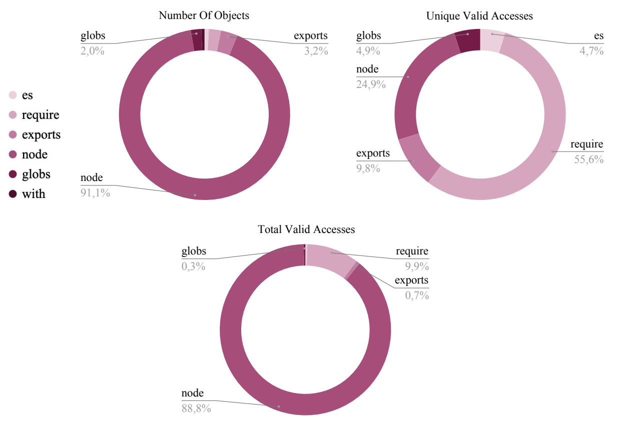


Fig. 7: Access characteristics as a function of context. Clockwise from top-left: (i) number of LYA wrappers, (ii) unique valid accesses (*i.e.*, counting each access once), (iii) total valid accesses (*Cf.*§5.3).

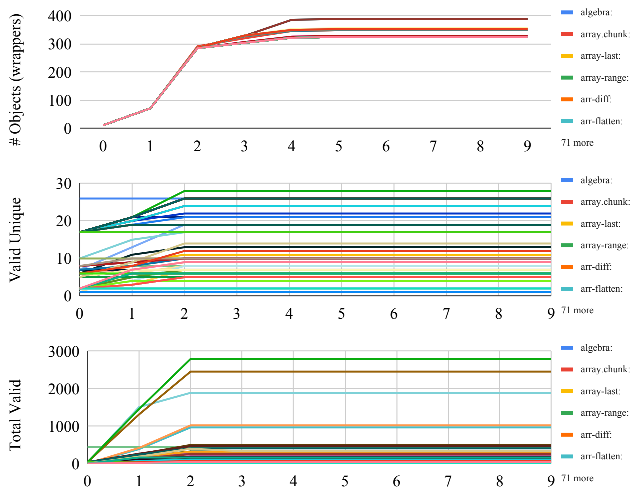


Fig. 8: Analysis characteristics as a function of the analysis depth. From the top: (i) number of LYA wrappers applied, (ii) number of unique accesses (*i.e.*, counting each access once), (iii) total number of accesses (*Cf.*§5.3).

Fig. 8 shows LYA's context characteristics on all 50 libraries (§5.1). 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 `exports`.

6 Discussion

This section discusses LYA's minimum requirements (§6.1) and reflects on its strengths and weaknesses (§6.2).

6.1 Applying LYA to Other Environments

The techniques presented in this paper are applicable to any programming language. We chose JavaScript primarily because (i) it boasts the largest collection of libraries [7] and (ii) we had prior experience developing more sophisticated versions of SA and PD. We now expand on the requirements for bolting our techniques on other runtime environments.

By operating on library boundaries, LYA uses a few special language features supported by all dynamic programming languages. First, LYA needs to re-wire the `import` keyword: if it is a function (*e.g.*, JavaScript, Scheme, Racket), re-wiring amounts to an overwrite; if it is a special non-functional keyword, re-wiring needs to be prefixed by a rewriting pass replacing `import` with a function (during module loading).

It also needs the ability to interpose on values and transform their components. Dynamic programming languages offer runtime reflection and conveniently expose object accesses as (over-loadable) functions—*e.g.*, `Proxy` objects in JavaScript, metatables in Lua, metaclasses in Python, and direct-accessor meta-programming Ruby. Compiled languages would apply transformations with static metaprogramming facilities, traversing and transforming objects at compile time. Examples include templates and macro expansions such as Template Haskell and Rust's macro system.

LYA makes extensive use of hash-maps to check for object equality, for cycles during traversal, and for prior wrapping. Hash-maps are supported by every programming language; many support storing weak references, which avoid affecting object reclamation in garbage-collected environments, and can be created either explicitly (*e.g.*, Python) or implicitly by using a "weak" structure (*e.g.*, JavaScript).

Finally, LYA makes use of variable shadowing to hide the original, unmodified versions of values—a universal feature.[4]

It is worth noting that LYA does not require the entire source code of the library to be available. While this is common in dynamic languages, LYA can operate on language-aware wrappers as is common, for example, with Python wrappers around C libraries.

To conclude, all major languages provide facilities to support a LYA-like framework, albeit with a different level of convenience. Dynamic languages have features—*e.g.*, name (re-)binding, value introspection, dynamic code evaluation, and access interposition—that enable runtime transformations, which conveniently unify module identification with interposition: a single function-like operator locates a module, interprets it, and applies transformations before exposing its interface in the caller context. Static languages have other benefits, such as avoiding the runtime overheads of transformations by applying them at compile-time.

6.2 When to Use LYA-like Analysis

While heavy-weight dynamic analysis provides full visibility into program execution, it may suffer from higher performance overheads, require significant developer effort, or offer results that are too low-level for the problem at hand.

[4]A notable exception is the language CoffeeScript.

The set of techniques presented in this paper shines at a need for urgency, select online analysis, and a high level of abstraction. It can help identify quickly the source of a problem by swiftly isolating homegrown code from standard and third-party libraries, by requiring a few lines of analysis-specific code to be inlined, and by leveraging the developer's expertise in their language of choice. By operating at coarser granularity, using the vanilla optimized runtime, and by toggling parts of the analysis on and off, it provides the ability to perform the analysis online. Finally, it deconstructs programs only at library boundaries, a naturally coarser granularity that is ideal for certain classes of problems.

LYA's set of techniques is *not appropriate* for monolithic programs or ones written in low-level languages such as C, due to several reasons. C neither features import/export keywords and runtime code evaluation, nor clear, "impenetrable" boundaries like the ones available in memory- and type-safe languages. C libraries can forge pointers and access arbitrary locations in the program's address space, a feature that breaks a key hypothesis in LYA—namely, that the wrapping mechanism can track a library's full observable behavior. A final problem is C's inability to traverse and wrap objects; due to lack of runtime information about size and structure, it is unclear how to traverse and transform them.[5]

7 Related Work

The previous section (§6.2) offers a theoretical comparison between conventional dynamic analysis and LYA. This section we turn to a technical comparison of LYA to prior systems.

There are several dynamic analysis frameworks for JavaScript [2,18,30,40,44]. Jalangi [40] and Mugshot [30] are the most popular and feature fine-grained code-level instrumentation powerful enough to record and replay entire program executions. This power comes at a significant inconvenience: they incur 2–3 order-of-magnitude slowdown and are not embeddable in a runtime-agnostic manner. These features are antithetical to LYA, which leverages basic programming-language primitives to insert high-performance analysis hooks completely transparently to the runtime environment.

NodeProf [44] is a fine-grained dynamic analysis standing at a different point in the analysis design space—more powerful than LYA, but with a significantly higher effort to use. It instruments a program's AST (rather than the code) statically, limiting the use of dynamic features, assuming knowledge of the underlying Graal [50] and Truffle [51] APIs, and requiring recompilation of the Java hooks. Nevertheless, some of NodeProf's goals are similar to LYA such as lowering overheads and toggling analysis during execution.

The Aran sketch [2] proposes proxy-based analysis coupled with JavaScript's unusual `with` primitive for wrapping custom contexts. LYA works at the module boundaries, a potentially coarser granularity, without any unusual language constructs.

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 [22], 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 heavyweight instrumentation.

LYA has some resemblance to dynamic instrumentation frameworks [13, 26, 28, 32]. These wrap basic blocks of a program incrementally and right before execution, similar in vein to how LYA wraps libraries. However, they operate at a much lower level (binary) than LYA, are much more detailed and heavy-weight, and are usually not available to high-level languages as a language-aware library.

Aspect-oriented programming (AOP) is a programming model where program points map to actions taken at these points [19]. A part of LYA is genetically related to AOP, drawing from proxies, wrappers, mirrors and other related mechanisms up to metaobjects [20]. While aspects are more powerful than LYA's localized transformations, adding support for them in a language is not trivial: for example, AspectJ and AspectC++, while extensions of popular languages, require modifications to the original language and/or runtime system. LYA operates on a completely unmodified language and runtime environment.

LYA draws inspiration directly from program fracture and recombination (PFR) [1,41], 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 libraries or modules.

8 Conclusion

Recent trends in software engineering have led to unprecedented levels of third-party code (re-)use. We present LYA, a novel library-oriented dynamic analysis framework for dismantling, analyzing, and reassembling programs by combining name shadowing, context wrapping, and transformation of the underlying dependency graph. LYA delivers better performance than conventional analysis systems, improved compatibility, and support for a range of dynamic analyses.

Acknowledgments

We thank Ben Karel, Jeff Perkins, and the MIT PAC group for their valuable thoughts. Much of LYA's design was informed though interactions with the community—we are particularly thankful to Isaac Z. Schlueter and CJ Silverio from

[5]Recent efforts such as C-Strider [39] and ptrSplit [23] show promise.

npm and Daniel Katz, Petros Efstathopoulos, Kevin Roundy, and Daniel Marino from Symantec/NortonLifeLock Research Group. This work was partly supported by DARPA contract no. HR00112020013 (ALRSCC).

References

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

[2] Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. Poster: Dynamic analysis using javascript proxies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 813–814, Piscataway, NJ, USA, 2015. IEEE Press.

[3] Russ Cox. Surviving software dependencies. *Commun. ACM*, 62(9):36–43, August 2019.

[4] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, page 3, USA, 2003. USENIX Association.

[5] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 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*, SEC'18, page 343–359, USA, 2018. USENIX Association.

[6] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

[7] Erik DeBill. Module counts. http://modulecounts.com, 2009. Accessed: 2020-01-11.

[8] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, page 214–233, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] Thomas "Halvar Flake" Dullien. Security, moore's law, and the anomaly of cheap complexity. http://rule11.tech/papers/2018-complexitysecuritysec-dullien.pdf, 2018.

[10] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, page 35, New York, NY, USA, 2004. Association for Computing Machinery.

[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, December 2007.

[12] Mohammad Fares. Record your terminal and generate animated gif images or share a web player. https://github.com/faressoft/terminalizer, 2019. Accessed: 2020-01-14.

[13] Cormac Flanagan and Stephen N. Freund. 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*, PASTE '10, page 1–8, New York, NY, USA, 2010. Association for Computing Machinery.

[14] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[15] Ev Haus. Jest performance is at best 2× slower than jasmine, in our case 7× slower. https://github.com/facebook/jest/issues/6694, 2016.

[16] TJ Holowaychuk. Expressive middleware for node.js using es2017 async functions. https://github.com/koajs/koa, 2014. Accessed: 2020-01-14.

[17] Hrishikesh. Dockerhub: Jalangi docker container. https://hub.docker.com/r/hrishikeshrt/jalangi, 2018.

[18] Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using javascript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 49–60, New York, NY, USA, 2013. ACM.

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

[20] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[21] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. 2017.

[22] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages*

and Operating Systems, ASPLOS '19, page 1045–1058, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2359–2371, New York, NY, USA, 2017. ACM.

[24] SS Jeremy Long. Owasp dependency check. 2015.

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

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[27] Michael Maass. *A Theory and Tools for Applying Sandboxes Effectively*. PhD thesis, Carnegie Mellon University, 2016.

[28] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: A domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 239–250, New York, NY, USA, 2012. Association for Computing Machinery.

[29] Bertrand Meyer. Design by contract: Making object-oriented programs that work. In *Proceedings of the Technology of Object-Oriented Languages and Systems - Tools-25*, TOOLS '97, page 360, USA, 1997. IEEE Computer Society.

[30] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 11, USA, 2010. USENIX Association.

[31] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by Contract, by Example*. Addison Wesley Longman Publishing Co., Inc., USA, 2001.

[32] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

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

[34] Inc. npm. Compromised version of eslint-scope published. https://status.npmjs.org/incidents/dn7c1fgrr7ng, 2018. Accessed: 2018-07-12.

[35] npm, Inc. Details about the event-stream incident. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident, 2018. Accessed: 2018-12-18.

[36] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, page 18, USA, 2003. USENIX Association.

[37] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery.

[38] Sam Saccone. npm fails to restrict the actions of malicious npm packages. https://www.kb.cert.org/vuls/id/319816, 2016.

[39] Karla Saur, Michael Hicks, and Jeffrey S. Foster. C-strider: Type-aware heap traversal for c. *Softw. Pract. Exper.*, 46(6):767–788, June 2016.

[40] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon 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*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[41] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not.*, 50(6):43–54, June 2015.

[42] Snyk. Find, fix and monitor for known vulnerabilities in node.js and ruby packages. https://snyk.io/, 2016.

[43] Ayrton Sparling et al. Event-stream, github issue 116: I don't know what to say. https://github.com/dominictarr/event-stream/issues/116, 2018. Accessed: 2018-12-18.

[44] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 196–206, New York, NY, USA, 2018. ACM.

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

[46] Emerson Thompson. A foreign exchange rates and currency conversion using cli. https://github.com/thompsonemerson/moeda, 2017. Accessed: 2020-01-14.

[47] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. Ignis: Scaling distribution-oblivious systems with light-touch distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1010–1026, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Shiyi Wei. Blended analysis for javascript: A practical framework to analyze dynamic features. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 101–102, New York, NY, USA, 2012. Association for Computing Machinery.

[49] Ashley G Williams. Changes to npm's unpublish policy. http://blog.npmjs.org/post/141905368000/changes-to-npms-unpublish-policy, 2016.

[50] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöundefined, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.*, 52(6):662–676, June 2017.

[51] Thomas Würthinger, Christian Wimmer, Andreas Wöundefined, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

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

[53] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 995–1010, USA, 2019. USENIX Association.