Iris: Language-Based Module-Level Compartmentalization

Anonymous Author(s)

Abstract

Module-level compartmentalization (MLC) is a popular technique for isolating program modules from each other and the surrounding environment. We present a new compartmentalization approach that targets modern dynamic languages, such as JavaScript, enabled by the fact that they feature a module-import mechanism that loads code at runtime as a string. The 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 security-specific code into the module before it is loaded. This code implements the compartment boundary, virtualizing the full set of names that refer to values outside the compartment. Experiments with IRIS, our prototype targeting the JavaScript ecosystem, demonstrate significant benefits in terms of compatibility, manual effort, protection granularity, and order-of-magnitude improvements in runtime performance over state-of-the-art MLC.

1 Introduction

Compartmentalization Compartmentalization is the decomposition of a program into multiple isolated components, termed *compartments*, aimed at improving its security: compartments collaborate to fulfill the program's goal but are guarded from accessing each other and the surrounding environment. Recent work on system-based module-level compartmentalization (SMLC) [31, 49, 66, 68] demonstrates its value against software supply-chain attacks [11, 32, 35, 36, 61, 67]. By offloading protection to the operating system (OS), SMLC protects against a realistic threat model that includes defective, subverted, and actively malicious modules.

Despite these benefits, SMLC is hampered by several practical limitations: the introduction of concurrently executing compartments may break the program's behavior, i.e., introduce *incompatibilities*; synchronous, blocking module interfaces at the compartment boundary may need to be *manually rewritten* to non-blocking ones; the *granularity* of privilege control is often coarse (*i.e.*, at the level of entire modules or system calls); and the use of heavyweight isolation mechanisms introduces significant *performance overheads*.

Language-based Compartmentalization To address these limitations, we present a new language-based module-level compartmentalization approach (LMLC), targeting modern dynamic languages and prototyped in a tool called IRIS. LMLC's insight is that, as long as certain conditions hold, most of SMLC's goals can be achieved by applying language-based techniques from within a compartment (rather than applying OS-based techniques around it). For example, in place of conventional SMLC guarding a compartment `a` from reading the process environment, writing to the file-system, and executing a function `foo` from a different compartment `b`, LMLC disables `a`'s access to `process.args`, `fs.write`, and `c.foo`—all by operating on names defined in `a`'s scope. This is achieved using a combination of load-time source code transformations that operate on the string representation of a module, as well as dynamic transformations that operate on the context to which it is about to be bound. Combined, these transformations insert security-specific code into a module before it is loaded, creating a virtual compartment that separates the module from its environment.

To configure the interaction between compartments, LMLC provides a set of abstractions inspired by the latent contract literature [7, 18, 19, 22]. IRIS's contracts are expressed in (a subset of) the same language as the programs they protect, using the same abstractions and built-in libraries. Using the LMLC's aforementioned mechanisms, contracts are protected in a separate compartment and have various built-in interfaces disabled to avoid breaking compatibility.

Key Results In cases where LMLC can be used (see §3 for the threat model), it promises significant benefits: it overcomes SMLC's compatibility issues; it is successful at mitigating both real and hypothetical attacks; it supports security contracts that are highly expressive (> 500 fields per compartment) and relatively concise (<100 LoC), by using powerful language constructs; and delivers up to three orders of magni-

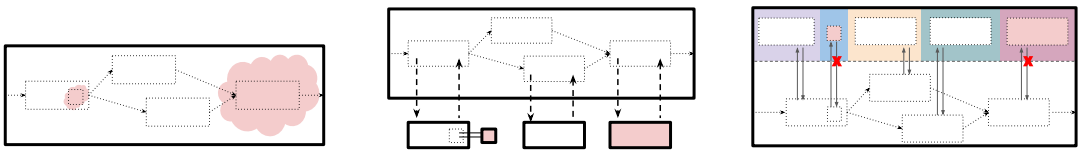

Fig. 1: Overview of IRIS Left: an application with many third-party modules, some of which may be malicious (*Cf.*§2.1). Middle: conventional, OS-based module-level compartmentalization (*Cf.*§2.2). Right: language-based module-level compartmentalization with IRIS (*Cf.*§2.3).

tude performance improvements over state-of-the-art SMLC systems, with compartment startup and runtime overheads remaining under 1ms and 7%, respectively.

State of the Art IRIS shares characteristics with prior work on dynamic wrapping, ranging from Lisp and Smalltalk [9, 21, 28, 60] to JavaScript [3, 14, 42–44], but differs in both its focus and technique. Its focus is multi-compartment isolation, thus control the full set of names observable from within a compartment such as `Error.captureStackTrace` and `process.args[0]`, rather than call-argument wrapping at the boundary. To achieve this, among other differences, its techniques apply source code transformations prior to wrapping. (§8 offers more details.)

Structure & Contributions The paper starts with an example of third-party library concerns, the challenges of SMLC, and an overview of LMLC (§2). It then discusses IRIS's threat model (§3). Sections 4–7 present our key contributions:

- A set of fine-grained compartment specification abstractions for controlling inter-compartment access (§4).
- A set of static and dynamic transformations applied over modules and their context to create compartments (§5).
- An open-source implementation (§6) as a language module, along with its performance and security evaluation (§7).

While problems with third-party modules are pervasive, IRIS's implementation and evaluation focus on the JavaScript ecosystem as (i) it has the largest collection of modules [15] (and number of problems [20, 75]), and (ii) it simplifies comparison with prior MLC systems [68]. The paper closes with a discussion of necessary conditions for applying LMLC, based on our experiences developing IRIS (§9), and a detailed comparison with prior work (§8).

Non-contributions & Limitations Following the separation of mechanism and policy in the operating-systems [33] and compartmentalization literature [6, 10, 12], IRIS's goal is to provide a contract-parameterizable security mechanism. We expect that creating contracts will be the target of automation—*e.g.*, static analysis, dynamic analysis, domain-specific languages, or a combination thereof. Thus, we do not make claims about the simplicity of the contract language, but do report on our experience using it. Future work can focus automatically creating contracts for IRIS, building on its high-performance implementation platform.

2 Background and Overview

We illustrate the problems of third-party modules (§2.1) by revisiting a large-scale compromise, introduce conventional MLC and its limitations (§2.2), and close with an overview of IRIS (§2.3). The discussion is accompanied by Fig. 1.

2.1 Running Example: A Bitcoin Wallet

To understand the problems with third-party libraries, consider the recent `event-stream` incident [50, 62]. Meant to simplify working with data-streams, at the time of the incident `event-stream` was used by hundreds of applications and averaged about two million downloads per week. At one point, its author handed off maintenance to a volunteer; this is common practice when an open source developer reaches a saturation point. The new maintainer added an obfuscated, malicious package as a dependency to `event-stream`, called `flatmap-stream`, with code that was designed to harvest account details from selected Bitcoin wallets.

Fig. 2 zooms into (simplified) fragments of the attack. If run as part of a specific Bitcoin application (Copay), it starts by loading the `account.js` module related to the credentials of the user's Bitcoin wallet (line 4). Since the module is already loaded by the runtime, the language's module system returns a cached copy of the module's return value. This gives the attacker the ability to overwrite (*i.e.*, monkey-patch) its `getKeys` method at runtime (l.5,6). The new method accesses the sensitive account credentials by invoking the original method (l.7). It then loads the `http` module, and transmits the credentials to a remote, third-party server (l.8). Finally, it returns the expected results to the caller method (l.9).

While this is a simplified version of the attack, used to illustrate MLC and IRIS, *the real attack is not detectable via static or dynamic analysis*. Static analysis would not have helped because the attacker employed a series of encryption passes over the malicious code [58]. Dynamic analysis would not have helped either, because the malicious code activated very selectively: only when `event-stream` was part of Copay's dependency tree, only when run on the "live" bitcoin network, and only on users that had a balance of ฿100 or more.

This case is not alone: modules have exfiltrated sensitive environment variables [8], SSH keys [51], and credit card numbers [52]. While the JavaScript ecosystem dominates headlines—with >1M packages, >150K authors, >1B downloads per day—the trends are widespread across languages

and worsening: a record-setting 16K new vulnerabilities due to third-party packages were disclosed in 2018 [63].

2.2 Conventional, OS-enforced MLC

The key issue underlying the `event-stream` attack is that any third-party fragment of an application has unrestricted access to the functionality available to the rest of application. Some of this functionality is *explicitly* provided by other libraries such as `fs` and `http`. Other functionality is *implicitly* provided by the programming language; examples include the ability to use global variables, import modules, and access the cache of the loaded modules. Both explicitly and implicitly provided functionality is exploitable by third-party code. While it may be needed for the application to function as a whole, it is not necessarily needed by `event-stream`.

OS-enforced MLC (SMLC) [31, 49, 68] leverages this insight to restrict functionality at the boundaries of third-party modules. For example, SMLC would protect against `event-stream` and its malicious dependency by spawning them as separate processes. Address space isolation makes top-level and global objects of the application inaccessible; process containment restricts the module's access to the file system and the network. and local copies of interfaces localize the effects of overwriting module APIs. SMLC is a powerful tool for security; however, it creates a few challenges of its own (**C1–4**).

Compatibility (C1) SMLC can introduce *incompatibilities* by creating concurrently executing compartments [10, 25, 68]. Ensuring that the compartmentalized program preserved the behavior of the original program is challenging; after all, introducing concurrency adds behaviors that do not exist in the original program. Examples include new sources of non-determinism, multiple copies of code and data, and conversion of cooperative scheduling to preemptive. Access to state shared among several modules can lead to inconsistencies, race conditions, deadlocks, causal ordering violations, and other classic distributed-systems problems [25, §2.4]. Execution must also deal with the potential for compartment failures, which in turn can also compromise assumptions about execution behavior and global invariants.

event-stream

```
let es = exports;
es.map = require("flatmap-stream");
es.pause = ...
```

flatmap-stream

```
let s = require("stream");
exports = (m, o) => {...}
...
let c = require("btc-wallet/account.js");
let gk_old = c.getKeys;
c.getKeys = (...args) => {
 let k = gk_old(args);
 require("http").request(RMT_SRV).end(k);
 return k;
}
```

Fig. 2: Use of third-party modules. Malicious code, highlighted in red, patches the `getKeys` method, spoofing credentials and sending them over the network to a remote server.

Manual Effort (C2) In many cases, SMLC implicitly alters the programming model at the module boundary, which itself requires significant *manual effort*: as I/O is introduced, module interfaces that are blocking often need to be converted to ones that are non-blocking, asynchronous, and concurrent. When a function evolves from being compute-only to potentially yielding, all functions along the path from the function whose call semantics have changed up to the root of the call graph may potentially have to be split in two—an effect known as "function ripping" [2].

Granularity (C3) SMLC is often restricted to *coarse-grained* monitoring and enforcement, because OS mechanisms are applied over coarse boundaries—such as processes *etc.*—rather than fine ones (*e.g.*, individual object fields or methods). It can easily allow or deny access to an entire module, and often monitor OS-level interfaces—with mechanisms such as system-call interpositioning and container namespacing; however, semantic monitoring at the granularity of module fields requires support from inside the compartment, which is outside the strict visibility of OS mechanisms.

Performance (C4) Most importantly, SMLC incurs significant *performance overheads* due to heavyweight OS mechanisms enforcing boundaries across the entire dependency graph. Invocations at the boundary become excessively costly due to (de-)serialization, interprocess communication, and context switching costs—especially problematic in cyclic dependencies where a boundary has to be crossed multiple times for a single top-level call. Spawning copies of the runtime system, built-in libraries, and the compartmentalization framework for every new compartment requires vastly more resources (*e.g.*, memory, file descriptors, buffering) than shared-address-space modules. When deployed at scale, all these overheads also translate to prohibitive financial costs.

Prior work attempted to ameliorate the effects of these challenges—*e.g.*, tackling **C1** with concurrency control [26, 71] and **C4** with hardware acceleration [12, 17] (more in §8). But while these challenges are inherent to SMLC and have to be explicitly dealt with, they are non-existent in LMLC.

2.3 Language-based MLC with Iris

Iris takes a different approach, by virtualizing the full observable environment around a module—*i.e.*, not just the interface of the module and importing modules, but also names that are made available by the language and resolve to values outside that module. Rather than relying on the operating system to protect memory accesses, it shadows variable names from within a compartment controlling its ability to name disallowed functionality. Shadowing creates a new scope that

redefines variables available in the outer scope, forcing a compartment to resolve all names in this new scope. A module thinks it is accessing original values, when in fact it is accessing IRIS-provided values that verify accesses before propagating them to the original values. For example, the `event-stream` module cannot bypass IRIS to name the application's top-level and global objects, cannot access the original functions of `fs` and `http`, and cannot overwrite the `getKeys` field of a read-only `wallet`.

To restrict the privilege available to compartments, IRIS allows the specification of privilege contracts (PICs) operating at the compartment boundary. PICs take the form of function predicates, Turing-complete monitoring functions controlling access to names that are defined outside a boundary. PICs operate at a very fine granularity—that of individual fields of individual objects that form the context of individual modules. For example, a PIC can express that the `event-stream` module can use the built-in `require` function as long as it only imports `console` (for writing to the standard output stream).

We discuss PICs after a careful overview of the threat model.

3 Threat Model

IRIS' goal is not to deal comprehensively with the intricacies of dynamic languages, but only to provide a refined version of compartmentalization—*i.e.*, one that addresses **C1–4**.

Threats and Goals Broadly, IRIS protects against (i) buggy modules subverted at runtime by attacker input that triggers unintended actions (*e.g.*, call a method to access a secret key); (ii) malicious modules indirectly coercing other modules into performing unintended actions (*e.g.*, use introspection to get a secret key); and (iii) malicious modules directly attempting unintended actions (*e.g.*, exfiltrate a secret key directly).

IRIS aims to disable "ambient" overprivilege by allowing users to create lightweight compartments, controlling module-to-module access at a *very fine* granularity—that of individual names, functions, and fields of individual module return values. Similar to the SMLC literature, IRIS aims at controlling a compartment's full observable behavior: there should be no name accessible from a compartment that is not monitored by IRIS. This includes built-in libraries, such as `fs` and `net`, language features, such as call stack inspection, prototype accessing, and runtime reflection, and system-based resources, such as `process.env` or `process.args`. Informally, as long as an interface is accessed via a name that resolves to outside the library, the access will hit IRIS.

Assumptions/Limitations IRIS places trust in the language runtime and built-in modules such as `fs`. Although it can confine built-in modules, a minimum of trusted functionality is needed from the module system to locate and load contracts. Moreover, when IRIS is introduced as a module, it is assumed to be loaded *before* any other module—otherwise, a malicious module could dynamically rewrite IRIS's code.

Tab. 1: Implicit imports. Examples names resolving to a scope outside that of a module come from (i) core built-in objects, (ii) the standard library, (iii) implementation-specific objects, (iv) module-locals, and (v) global variables.

Source	Example Variables
Built-in Objects	`Object`, `Function`, `Array`, `eval`, `parseInt`, ...
Standard Library	`Math`, `Number`, `String`, `JSON`, `Reflect`, ...
Node globals	`Buffer`, `Process`, `console`, `setTimeout`, ...
Module locals	`require`, `_dirname`, `exports`, `_filename`, ...
Globals	`GLOBAL`, `global`, `Window`, ...

IRIS aims at providing (a refined version of) compartmentalization, which is fundamentally about controlling access to interfaces—including built-in or system interfaces like the ones mentioned earlier. This differs from more sophisticated language-based work that, for example, provides deep attenuation, membrane-like wrapping of function arguments [46], or information flow tracking.

Characteristics	S	L
Compatibility	−	+
(Finer) Granularity	−	+
Rewriting Effort	−	+
Runtime Performance	−	+
Protect Native Code	+	−
Scale out against DoS	+	−

Contrary to SMLC, IRIS does not handle native modules and denial-of-service (DoS) attacks. Native modules are written in lower-level languages such as C, C++, or assembly, and can bypass any security guarantees provided by the higher-level programming language and enforced by the runtime; any operation that violates memory safety (*e.g.*, pointer creation) would allow arbitrary modules to bypass IRIS's language-based protection techniques. DoS attacks attempt to prevent legitimate use of a module by overloading it (*e.g.*, sending many requests, or few carefully crafted ones; infinite loops); weakening of DoS attacks by automatically scaling out would not work in IRIS's single-process environment. For these two classes, developers can use (i) OS-enforced MLC systems [10, 68] and pay the aforementioned costs for a small fraction of the code base, or (ii) a system targeting that particular class (*e.g.*, NaCl [74] for native code; DeDos [16] for DoS attacks).

4 Interface Privilege Contracts

As a first step, IRIS arms developers with the ability to explicitly specify isolated compartments, by specifying subsets of rights granted to modules by default upon import. This ability is available on a per-import basis, through each module's so-called interface privilege contract (PIC). A key observation is that functionality accessible by default in a compartment can be modeled provided by implicitly imported modules.

Explicit *vs.* Implicit Imports IRIS unifies two different (and broad) classes of mechanisms that modules can use to import functionality. Functionality available to modules is either

(i) *explicitly* imported from other modules, or (ii) *implicitly* available through their context.

Functionality can be imported with the use of an explicit `import` or `require` statement. Such a statement returns a value—the general case of which is an object—that becomes available to the module's scope. For example, a `log` module may return an object with method fields `info`, `warn`, and `err`, of which only `info` might be needed.

Functionality can also be available by default through (and shared with) the module's "outer" *context*. Examples of such implicit functionality (Tab. 1) include top-level objects and functions (*e.g.*, `process.args`, `eval`), functions to output messages (*e.g.*, `console.log`), and capabilities related to module-importing *itself* (*e.g.*, `require`, `exports`). Naming this functionality resolves to a scope outside that of a module, and is pervasively accessible from any point in the code. What ends up hitting such an implicit context depends on the language's variable name resolution[1] and represents ambient authority [45]: a module need only specify the names of the objects and methods associated with an operation in order to invoke it. Similar to explicit imports, implicit ones are accessible through the module's return value.

Specification Granularity A PIC restricts the privilege available to a compartment c (one or more modules) by refining privilege across all the fields across all import values accessible by c. This refinement is expressed via predicates that need to hold for an access to succeed; semantically, an access is allowed only if a predicate evaluates to `true` when the access is attempted at runtime.

To make it easier to describe PICs and associated transformations (§5), PICs follow the distinction between explicit and implicit imports (Fig. 3). The part of a PIC targeting explicit imports is termed the *explicit segment*, and defines how a module's interface is used by its consumers. The part of a PIC targeting implicit imports is termed the *implicit segment*, and defines what implicit functionality it imports. A module's explicit segment may be different every time the module is imported, whereas its implicit segment does not generally change. Examples of PIC segments include "can only write the first element of an imported array" (explicit) and "can only read the `PWD` environment variable" (implicit).

The combination of modules and PICs results in a new mental model, in which developers think of `import` as an augmented operation that supports privilege specification. Upon import, they can attenuate the privilege of individual modules by making privilege explicit at a very fine granularity. Under this extended model, IRIS can be thought as an "object-path" protection service: access rights are expressed as privilege associated with a path from the program's module roots to the field currently being accessed. While implicit imports are part of the model, module-local and function-local values are *not*: IRIS does not allow specifying (or enforcing) access restrictions on, say, arbitrary objects or function return values. Whenever such restrictions are needed, values have to be wrapped in and imported from standalone modules.

[1] For example, in some cases variables need to be prefixed with `global` (*e.g.*, Python) while in others resolution is attempted on increasingly outer scopes until it hits the global scope (*e.g.*, JavaScript).

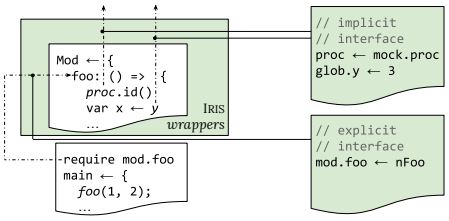

Fig. 3: Contract segments. Cross-module variable name resolution (left) augmented with IRIS (green boxes), which interjects non-bypassable steps resolving to IRIS-augmented values (right-top: implicit module imports; right-bottom: explicit import).

PIC Expressions PICs can be expressed using a subset of the source language, enough to map aforementioned object-paths of a module's return value to access predicates. They can be specified directly at the point where a module is imported (*e.g.*, `require("ev-stream", σ)`) or as part of an auxiliary PIC contract file (*e.g.*, `pic.json`). The latter is intended to simplify interaction with automated tools built atop IRIS.

Privilege refinement is expressed via function predicates over the type of the attempted access. They take as input the values associated with the access and return a boolean, `true` for allowing access to go through and `false` for denying access. Input parameters to predicate functions for values that are read, written, and deleted include (i) a pointer to the target object, (ii) the name of the field to be accessed, and (iii) for writes, the new value. For executable values and constructors, input parameters include (i) a pointer to the target object (ii) the implicit "self" argument for method calls, and, most importantly, (iii) the list of arguments provided for the call. Predicate functions are Turing-complete, in that they can run programs of arbitrary complexity. In practice, however, they tend to be purely functional computations over the aforementioned inputs.

PICs work in conjunction with the existing language restrictions and protection mechanisms. The privilege associated with a value depends on the value's type. Specifically, it is meaningful only in the context of accesses related to that type, and can be thought as further constraining over the language's base types. Conversely, to retain the full capabilities of the language, a right to perform an access does not place any constraints on the type or form of the new value.

Example PICs To illustrate the use of PICs in practice, consider the Bitcoin application presented in §2.1 (Fig. 2).

The first privilege we might want to constrain is the use of `require`: simply granting a module all-or-nothing privileges over `require` unleashes too much power, as it then allows the module to import *any* module into the current scope. A PIC can constrain the import to *only*, say, `log`. This is accomplished with the following PIC for `event-stream`:

```
1 event-stream:
2   globals.require: (m) => m === "log"
3 [...]
```

During execution, the anonymous function on line 3 will be provided the arguments to `require`, of which it binds only the first—corresponding to the module's name. It returns true if it matches "log", allowing execution to proceed uninterrupted.

For a more advanced use, consider a predicate that only allows reads and writes to files that belong in a certain directory. Module `event-stream` is provided a privilege-refined version of `fs`'s `readFile` method that can absolute links starting with "/var". This is accomplished with the following PIC:[2]

```
1 event-stream:
2   fs.readFile: (f) => f.startsWith("/var")
3 [...]
```

The `resolve` method of the built-in `path` module returns an absolute pathname; subsequent prefix-matching returns a boolean value. Different libraries or compartments can be allowed to access disjoint parts of the file-system tree, which works because PICs specify privilege for individual contexts: a different module importing `fs`, say `log`, could restrict `fs` access to the local directory.

Subtleties The use of predicate functions could open several subversion vectors. To avoid this, IRIS restricts PIC expressions using techniques described in later in the paper (§5).

The first problem is that importing third-party modules in predicates could open the doors to malicious code within the PICs, foregoing any of IRIS's benefits. To solve this, IRIS disallows the use of `require` within PICs. Instead, by using context transformation and rebinding, it only permits access to built-in modules directly by *name*. For example, instead of `require("path")`, path's functionality is available directly through the `path` variable.

Even while restricting to built-in modules only, issues remain: asynchronous, non-blocking functions return before the results of the checks are made available. As a result, a sensitive resource can be accessed before the corresponding PIC has any time to be enforced. To solve this, IRIS allows calling only synchronous, blocking interfaces of the built-in modules. This is achieved by guarding the privilege to read non-blocking interfaces with a PIC that disallows access.

A last problem is related to PIC side-effects. PICs may need to mutate state outside the PIC; this is useful, for example, to enforce access to an interface only once or within a certain time frame. Moreover, state must be shareable between PICs to allow coordinated decisions across PICs. However, state shared between PIC- and application-resident code could lead to problems because malicious application code could attempt to violate confidentiality and integrity of state on which PICs depend (*i.e.*, bypassing the two prior techniques). To solve these problems, IRIS introduces a custom global context that is only available to PICs and isolated from the `global` context of the application. The map can be shared across PICs, and stores data that persist across executions of a single PIC.

The techniques outlined above (and detailed in the next section) effectively execute all PICs in a single compartment, unaffected by malicious application code attempting to read or write PIC-only state.

[2] An earlier version of this contract used `fs.resolve` to ensure a path is under `/var`, and was thus susceptible to a TOCTTOU attack similar to comparable SMLC policies—*e.g.*, syscall policies ([54] *vs.* [72]). As mentioned earlier (§1), creating contracts automatically should ameliorate these concerns.

Fig. 4: Transformation pipeline. Module loading is augmented with several stages that transform the module and its context before returning its value to the importing code. Stages (1), (3), and (5) manipulate in-memory objects; stages (2) and (4) operate on source code (*Cf.*§5).

5 PIC Monitoring Transformations

To reduce the privilege modules possess, IRIS needs to enforce the privilege granted to modules upon import by developers during the execution of the program. This is achieved by a series of automated runtime transformations that wrap modules with security monitors. Each monitor is responsible for overseeing accesses that cross a single module-to-module boundary and ensuring that they conform to the privilege specified by the accompanying PIC. If a violation is detected, IRIS throws a special exception that includes contextual information for diagnosing root cause.

To transform a module, IRIS augments several steps of the module loading process (Fig. 4) using a base transformation that wraps an object with a security monitor. It starts by intercepting calls to the `require` function, which locates the module's source code and PIC. It then constructs a custom context based on the PIC's implicit segment. It then encloses the module in a closure to leverage local variable resolution in order to link the custom context with the module. After source interpretation, IRIS transforms the resulting value by consulting the PIC's explicit segment, updates the IRIS-augmented module cache, and returns the value to the consumer. The following subsections detail these steps.

```
wrap (e: Value) : Value := match e with
  | {(s, v) :: vs}  →  {(s, wrap v) :: wrap vs}
  | [v :: vs]       →  [(wrap v) :: wrap vs]
  | λ(...args).f    →  λ(...args).{ σ(e, args)? f : () }
  | _               →  interpose(σ, e)
end
```

Fig. 5: 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*.

```
let Math = {
 add: (a, b) => {
  return a + b;
 },
 sub: (a, b) => {...},
 ...
 ...
}
```

```
let _M = Math, Math = {};
Math.add = (...args) => {
if (σ(_M.add, args)) {
 return _M.add(args);
else
 throw Iris.PrivException();
};
Math.sub = (...args) => {...};
...
```

Fig. 6: Example base transformation. Applying the base transformation to a `Math` object that contains an `add` method yields a new `Math` object. The new `add` is a closure over the old `add`. When called, it passes arguments to and invokes the old `add` if the original add should be called; otherwise, it throws an `ACException`.

Privilege Exceptions The semantics of the augmented model guarantee that a module field will be accessed only if the accessing module is allowed to do so and if the access conforms to the field's predicate; if not, the IRIS wrappers throw an access violation. To expose an access violation to the user, IRIS introduces a new exception, `PrivilegeException`, that bypasses the program's control flow to notify the user and provide contextual information. This information is intended to simplify diagnosis of the violation and its cause, a challenge compounded by IRIS's chosen domain—large dependency graphs with many third-party libraries of which the developer has little understanding. Specifically, the exception includes the type of violation, names of the modules involved, names of accessed functions and objects, and a stack trace.

While IRIS guarantees that unauthorized access will not be allowed, a malicious mediating module can still catch and silence an `PrivilegeException`. This does not violate the semantics of the privilege refinement model, but (at worst) may complicate debugging. To simplify this, IRIS can be configured to execute in a log-only mode, in which it logs exceptions (high I/O overhead), or a fail-stop mode, in which it prints information to the error stream and stops execution.

Base Transformation IRIS's transformations boil down to a base form `BT` that augments *objects* with runtime security monitors. At a high level, `BT` takes an object O and a PIC σ and returns a new object O'. Every field f of O is wrapped with a method f' defined to enclose f. At runtime, f' checks σ: if the access is allowed, it forwards the call to f; otherwise, it raises an exception.

To achieve this transformation, IRIS walks the object graph from the root and processes component values based on their types (Fig. 5). Rather than mutating original values, it copies them, applies transformations on the copies, and returns copies to the caller:

- *function* values are wrapped by a closure that forwards arguments to the original function if allowed to do so.
- *object* values are recursively transformed, with their getter and setter methods replaced similar to function values.
- *primitive* values are copied unmodified and wrapped with an access interposition mechanism.

Direct field accesses, such as assignments, require custom detection upon dereference. To achieve this, IRIS wraps fields with an interposition mechanism, essentially treating direct field accesses as function calls. Examples of such mechanisms include `Proxy` objects (JavaScript), metatables (Lua), metaclasses (Python), and direct-accessor metaprogramming (Ruby). IRIS's wrappers detect and record changes to any of the object's fields; nested wrappers monitor nested objects.

For further processing, IRIS maintains a handle to the root of both the unprocessed and the newly processed values. The unprocessed value is used to create objects that check for different PICs. The new value is used to revoke or alter privilege at runtime, a capability not explored further in this paper.

Fig. 6 shows the result of applying the base transform `BT` on simple `Math` object.

Import Interception IRIS's runtime enforcement component is introduced as a backward-compatible, drop-in replacement of the language runtime environment's *module system*—either as an application-specific module (*e.g.*, `iris` package) or bundled with a custom runtime replacing the language's module system for all applications (*e.g.*, IRIS-powered Node.js). It always starts by dynamically replacing `require`: instead of simply locating and loading a module, the function yields to IRIS.

IRIS first checks if the module has already been loaded and if it has been loaded with the same PIC. If both are true, it recovers and returns the transformed module value. If the module was loaded with a different PIC, IRIS loads the appropriate PIC and applies a transformation pass on a cached copy of the module. Otherwise, IRIS first invokes the built-in module loader to locate the module.

The process of loading new modules includes a phase of reading the necessary files as *source code* and a phase of interpreting them, interspersed by applications of transformations. Reading files returns a string representation of the code; interpretation uses the language's runtime evaluation primitives to convert the code to an in-memory object. A series of transformations is applied before and after interpretation.

For PICs, this distinction is not as important, other than to note that PIC contract files are loaded as strings. The PIC contract language is embedded in the source language, thus IRIS makes use of the language's built-in evaluation primitives to interpret the PIC contract file. The result is identical to PICs specified as arguments to `require`.

Context Transformation IRIS needs to prepare a transformed copy of the module's outer context, a map from variable names to their values. To achieve this, it creates an auxiliary hash table mapping names to transformed values. Names correspond to implicit modules such as globals, built-ins, module-locals, *etc.* (Tab. 1). Transformed values are created by applying the base transformation `BT` over values in the context, as specified by the PIC's implicit segment (Fig. 7).

While user-defined global variables are stored in well-

```
let CONTEXT = {
require: null,      //module-local; null for now
process: Iris.BT(process, σ1),
setTimeout: Iris.BT(setTimeout, true),
Number: Iris.BT(Number),
Array: Iris.BT(Array),
// [...another 150 entries...]
}
```

Fig. 7: Creating a custom context. A custom context mapping variable names to values is created by applying the base transformation (Fig. 5) on individual values of the default context.

known locations,[3] traversing the global scope for built-in values is generally not possible. To solve this problem, IRIS collects such values by resolving well-known names hard-coded in a list; different lists exist for different environments and versions of the language. Using this list, IRIS creates a list of pointers to unmodified values.

Care must be taken with module-local variables, which refer to information associated with individual modules. These are accessible from anywhere within the scope of a module (similar to global variables), but each module refers to its own copy of these variables. Examples include the module's absolute filename, its `export`ed values, and whether the module is invoked as the application's `main` module. Attempting to access them directly from within IRIS's scope will fail subtly, as they will end up resolving to module-local values of IRIS *itself*—and specifically, the module within IRIS that is applying the transformation. IRIS solves this problem by deferring binding for later, and specifically from within the scope of the module. This process is detailed in the next section; for now IRIS leaves the entries empty.

Fig. 7 illustrates the context transformation for the `flatmap-stream` module (Fig. 2–bottom), based on the PICs defined in its implicit segment.

Module Enclosure IRIS needs to link the module with the newly transformed version of its context. To achieve this, it wraps the module with a closure that starts by redefining and enclosing global variable names as module-local ones (Fig. 8). The closure accepts as argument the customized context and assigns its entries to their respective variable names. It is arranged in a preamble comprised of assignments executing before everything else in the module. When the closure is applied to the customized context, the module's return value is recovered in a side-effectful manner (as in the unmodified module system) by reading the module's `exports` variable—that is, the closure does not end with an explicit `return`.

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" module-local values augmented with security monitors. As a result, even if new code is runtime-evaluated at a later point, it will still be constrained to modified values by the language's resolution algorithm. This runtime code-rewriting

[3] For example, `globals` in JavaScript and `_G` in Lua.

```
1 (cxt) => {
2  let require = Iris.BT(require, σ0);
3  let process = ctx.process;
4  let setTimeout = ctx.setTimeout;
5  let Number = ctx.Number;
6  let Array = ctx.Array;
7  // [...another 150 entries...]
8    let s = require("stream");
9    exports = (m, o) => {...}
10   (flatmap-stream code)
11 }
```

Fig. 8: Module enclosure. The original module (shaded lines 7–10) is wrapped with a function (lines 1–7 and 11) that (i) takes the custom context (Fig. 7) as a parameter, and (ii) shadows globals, language built-ins, and module-locals by re-defining them as function-locals (lines 2–7).

technique works because the module is still at a loading stage prior to interpretation, represented as a string.

Late-bound, module-local variables are the result of applying `BT` over variable names in the current scope, which is now bound to the correct module-local value.

Module Transformation and Return Returning the module's value to its consumer comprises interpreting the module, linking it with the custom context, and applying transformations to its return value (Fig. 9).

When interpreted, the source code returns (the in-memory representation of) a function. Passing the custom context to the function evaluates the module. During this phase the module may attempt to access global objects and cause side-effects, but all these are subject to the privilege granted to the module. The module can "see" only a limited subset of interfaces throughout its lifetime. Evaluation returns the module's exported values, similar to the vanilla module system.

Right before returning the exports to the consumer, IRIS runs the return value through a final application of `BT`, consulting the privilege specified by the PIC's explicit segment (Fig. 9, lines 1, 7). At the compartment boundary between `event-stream` and the imported `flatmap-stream`, only the `map` field should be accessible (Fig. 2, line 2). The transformation makes all other fields of the `flatmap-stream`'s return value inaccessible to the consumer module (`event-stream`).

Augmenting The Module Cache Applications may import the same module at different points of the dependency graph. For consistency and performance purposes, module systems

```
1 let compiled = [native].eval(enclosed);
2 let rewired = compiled(CONTEXT);
3 let bucket = require.cache["flatmap-stream"];
4 bucket[σ.implicit] = rewired;
5 let instance = Iris.BT(rewired, σ.explicit);
6 bucket[σ.implicit][σ.explicit] = rewired;
7 return instance;
```

Fig. 9: Module linking and export. After interpretation, the wrapped module (Fig. 8 is linked to the customized context (Fig. 7) by function application; the return value is cached before (4) and after (6) the final transformation responsible for attenuating the return value.

maintain a cache of loaded modules. When an already-loaded module is imported again, vanilla module systems return a cached reference to the return value of the original module. The new privilege-augmented model, in which the same module can be loaded multiple times with *different* privilege, requires augmenting the module cache and associated operations.

IRIS extends the cache with two additional levels. The three levels are respectively indexed by module identifiers (as before), implicit segments, and explicit segments. For each module, the second level contains a collection of entries corresponding to mostly-transformed modules, and the third level contains fully transformed modules. Mostly-transformed modules 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 enforce the explicit segment. The reason for splitting the two levels is that modules are usually governed by a single implicit segment but multiple explicit segments, one for each of its consumers. A context transformation is applied only a few times, whereas a return-value transformation is applied on every `require`. The second level also contains a special entry (indexed by `"_"`) for the non-transformed return value, so that subsequent transformations can skip loading the module from disk.

When a module is already loaded, IRIS indexes by implicit segment to retrieve the right module instance. It then applies the transformation required by the explicit segment. New modules are inserted into the second-level cache right after evaluation of the interpreted function (line 3 of Fig. 9).

6 Implementation

This section describes a concrete implementation of IRIS for JavaScript, targeting the Node.js ecosystem. IRIS is integrated into npm and available for setup under `@blindorg/iris`.

PIC contracts (§4) use JavaScript's object notation paired with syntactic sugar for accessing built-in modules. Such sugaring is achieved by wrapping PICs upon import with a function that resolves the names of built-in libraries (*e.g.*, `path`, `fs`) to objects modified to allow only synchronous methods. IRIS's runtime enforcement component (§5) dynamically modifies several functions from the built-in `Module` module. IRIS adds several steps. It augments the `require` module-local method to capture a PIC as a second parameter. It augments the internal `compile` method to apply (i) source-to-source transformations to the string representation of the library, and (ii) dynamic transformations and wrapping to surrounding context and return-interface (§5). It augments the cache to support storage and recovery of the unmodified, context-customized, and interface-attenuated versions of each module.

While working on IRIS, we found it useful to develop a small library for generating and manipulating sets of PICs across entire sub-graphs of the dependency graph. The library provides primitives for programmatically manipulating interposition points, as well as declarative options for configuring names available to compartments *en masse*. For example, predicates `include` (implying all other excluded) and `exclude` (implying all other included) affect three sets of options: (1) `context`, the set of names available to the compartment (Tab. 1), (2) `modules`, the set of modules part of a compartment (*e.g.*, `left-pad`), and (3) `fields`, the set of strings to be dynamically traversed (*e.g.*, `prototype` or `toString`). The library is about 100 LoC.

7 Evaluation

IRIS's primary hypothesis is that language-based protection mechanisms can address the challenges of conventional compartmentalization systems. This section confirms this hypothesis by evaluating (i) the performance overheads associated with runtime enforcement and how they compare with a state-of-the-art SMLC system; (ii) the security protection benefits obtained by the use of IRIS; (iii) the use of IRIS on large-scale, real-world applications. IRIS' evaluation shows that:

- IRIS's overheads related to startup time, memory consumption, and compartment boundary crossing are 1–3 orders of magnitude lower than a prior MLC system (**C4**, §7.1).
- IRIS's enforcement component protects against known and hypothetical attacks from buggy, subvertible, and actively malicious modules using fine-grained (**C3**) contracts (§7.2).
- IRIS makes it possible to run large, compartmentalized applications, without any compatibility issues (**C1**) nor any manual effort spent rewriting interfaces (**C2**) at the boundary (§7.4).

Experiments were run on a server with 512GB of memory and 80 Intel Xeon E7-8860 cores clocked at 2.27GHz. The software setup included Node.js v6.14.4, bundled with V8 v5.1.281.111, LibUV v1.16.1, and npm v3.10.10, atop a Linux kernel v4.4.0-134. For memory experiments, Node.js was launched with `-expose-gc` and `-log-gc` to allow triggering garbage collection and verifying it ran.

7.1 Performance Evaluation

How does IRIS*'s performance compare with conventional MLC systems?* To answer this question, we perform a series of experiments comparing IRIS to BREAKAPP [68]—a state-of-the-art compartmentalization framework that can place selected JavaScript libraries in process-isolated compartments. The micro-benchmarks comprise six modules of varying size and complexity (Tab. 2), selected based on workload diversity and compatibility with BREAKAPP. Two of these benchmarks, `lpad` and `nacl`, are tested with inputs of different sizes to better stress the overheads of serialization: a small input of 1B

Tab. 2: Characteristics of the benchmark modules. Columns are as follows: module shorthand (shown in Fig. 10), size in terms of lines of code (LoC), and overview of its internals (Notes) (*Cf.*§7.1).

	Module	LoC	Notes
1	`verbs`	29	constant string-to-string map (`ENUM`)
2	`lpad`	52	small, pure, string-padding function
3	`chalk`	145K	factory builder objects
4	`debug`	554K	variadic arguments, self-references
5	`ejs`	59K	DSL compiler, extensive test suite
6	`nacl`	94K	CPU-bound cryptographic processing

and larger one of 2KB (shown as -L). Results are averages over 1M repetitions, with a warm-up phase of 1K repetitions. We are interested in the following overheads (**C4**): compartment startup time, compartment memory consumption, and the overhead of crossing boundaries between compartments.

Startup Time The top plot of Fig. 10 compares the compartment startup time between IRIS and BREAKAPP. Each bar measures the time to complete a `require` call, which includes locating and transforming the module. Compared to BREAKAPP, IRIS improves compartment startup times by a factor of 12–168×: compartment creation is accelerated by two orders of magnitude for the heaviest modules such as `chalk`, and three orders of magnitude for lightweight modules such as `verbs` and `lpad`. BREAKAPP's additional overheads (beyond transformations) stem from setting up a TCP channel (10–20ms), serializing and shipping the return API to the consumer module (5–10ms), and launching a new process executing a copy of the Node.js runtime (80–120ms).

Memory Consumption The middle plot of Fig. 10 compares the base memory consumption of individual compartments between IRIS and BREAKAPP. Each bar shows the amount of heap memory used by a process, as reported by V8. The values for baseline and IRIS include only the heap use for the individual module imported. BREAKAPP's values account for the memory of the new compartment: a fresh copy of the Node.js runtime takes about 20MB; BREAKAPP adds another few MB, mostly coming from its dependencies to deal with serialization and multiple channel types. IRIS improves memory consumption overheads by a factor of 19–31×. Notably, in no cases did IRIS force V8 to request additional memory from the OS—even when a few large modules, such as `nacl` and `chalk`, end up doing this for the baseline. Put differently, while loading large modules may cause performance side-effects that are observable from outside the language runtime (*i.e.*, the OS), IRIS's overhead remains minimal on top of that.

Boundary-Crossing The bottom plot of Fig. 10 shows the cost of crossing a compartment boundary by calling into a module (includes function-invocation time). Generally, IRIS introduces between two and three orders-of-magnitude lower overheads than BREAKAPP. The reason is that BREAKAPP adds serialization, interprocess communication, context switching *etc.*, whereas IRIS relies only on

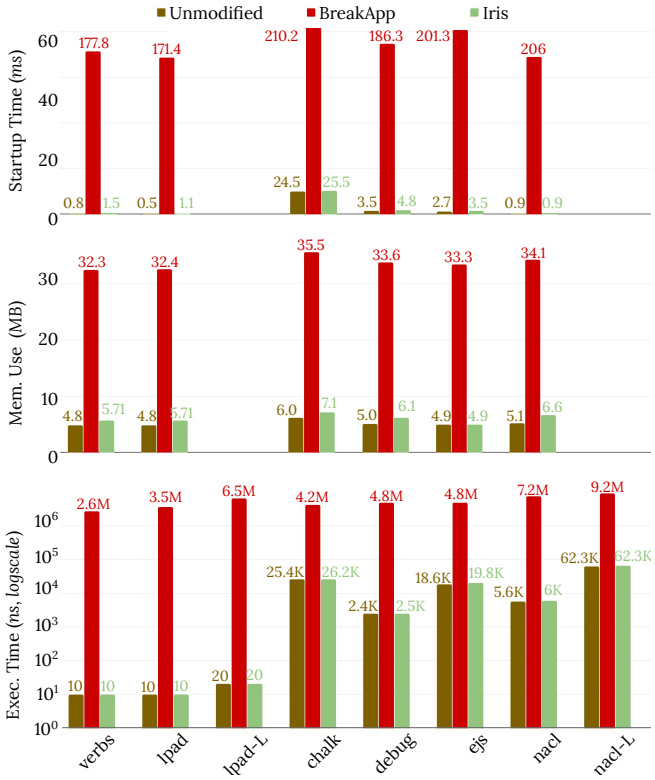

Fig. 10: Compartment overheads. Top: startup times. Middle: memory consumption. Bottom: boundary-crossing; as these overheads depend somewhat on the sizes of call arguments, `lpad` and `nacl` include runs with large values (-L) (*Cf.*§7.1).

language-level enforcement mechanisms.

Notably, IRIS's performance remains mostly unaffected by the input size (normal and `-L` columns). This is due to IRIS passing pointers rather than values between compartments: (i) copying a payload is significantly more expensive, even without considering BREAKAPP's user-kernel boundary crossing; (ii) serialization and deserialization transformations from in-memory values to strings and back. Further experiments (not shown) indicate that BREAKAPP's serialization overheads are dominated by value expansions due to cycle detection and property ownership. Thus, BREAKAPP's boundary-crossing cost is a function of the call-argument size.[4]

Take-Away IRIS *imposes minimal performance overhead, outperforming a state-of-the-art MLC system by several orders of magnitude.*

7.2 Security Evaluation

Does IRIS *succeed in mitigating vulnerabilities that fall under its threat model (§3)?* To answer this question, we perform experiments with real and synthetic vulnerabilities (Table 3).

[4] In theory, IRIS's overhead is a function of PIC size. In practice, however, PICs are only a few instructions long; being mostly constant values, they are optimized away by V8's tracing compiler.

Real-World Vulnerabilities The first batch of real-world vulnerabilities (rows 1–5) are drawn from widely-used modules in the Snyk database [61] and include the malicious versions and MITRE's common enumeration identifiers [47, 48]. We use the proof-of-concept exploit (PoCE) attached to the original vulnerability report. The `morgan` module is a request logger, which the PoCE exploits to call `console.log`; IRIS permits only importing and calling `morgan`. The `merge` module combines multiple objects into one, exploited by the PoCE to alter the object hierarchy with the injection of a `__proto__` property pointing to a new object parent; IRIS does not permit overwriting hidden properties, such as `__proto__`. The popular `mathjs` module includes a math evaluator, which the PoCE exploits to `console.log` a message; IRIS blocks it similar to `morgan`, not allowing use of `console`. The `ns` serialization module is exploited by the PoCE to import `child_process` and call `exec`; IRIS blocks this, by only allowing the `ns` import itself. The `st` module offers HTTP routing and caching that is used in conjunction with an HTTP server. It is susceptible to directory traversal: while dots (*e.g.*, `../`) are filtered out, url-encoded ones are not (*e.g.*, `%2e%2e/`). The PoCE attempts to access the file system root, but IRIS allows access only in the current directory.

We also apply IRIS on vulnerabilities found in the original BREAKAPP [68] (rows 6–10), some of which fall outside IRIS's threat model. We were not able to reproduce the attack on `fernet`. Six vulnerabilities (applied to `s/js`, `qs`, `hostr`) that fall within IRIS's threat model were mitigated, with relatively small PICs. By far the most interesting case was `hostr`, whose mitigation in BREAKAPP required the use Docker-style containment—BREAKAPP's most heavyweight isolation mechanism. IRIS mitigates access to the file-system by virtualizing its API through a check that applies `path.resolve` on the arguments. IRIS is not able to mitigate `uri-js`'s denial-of-service attack, in which a single call blocks V8's event loop indefinitely, nor `libxml`'s vulnerabilities; both sets of attacks fall outside IRIS's threat model.

Synthetic Vulnerabilities Tab. 3 also shows a *subset* of artificial attacks that we constructed to test IRIS, highlight unusual corner cases, and confirm our understanding. For this, we construct a module `mm` that exposes an `eval` function. Module `mm` lives in a separate, sealed compartment, and is fed strings that are evaluated at runtime.

We pass a large amount of code as string from IRIS's extensive testing infrastructure—over 800 tests (>11K possible accesses), targeting built-in JavaScript, Node, and Common.js interfaces—of which we only show five. Tests include unauthorized access of global variables, the module cache, the process arguments, and environment variables. IRIS allows primitive operations within `mm`, such as arithmetic on primitive values and basic string manipulation, the use of Array and Object literals, as well as the declaration and execution of functions in the scope of the compartment. However, it interposes on operations that have broader side-effects outside `mm`'s compartment. A few cases highlight IRIS' granularity aspects (**C3**): as PICs operate on individual fields, they are able to distinguish between different environment variables (`env.js`), process arguments (`arg.js`, and elements of imported arrays (`arr.js`).

Tab. 3: Vulnerable Modules A set of real (top) and synthesized (bottom) vulnerabilities mitigated by IRIS. PIC size is in number of lines (*Cf.*§7.2)

Module	Version	Attack Type	CVE	PIC LoC
`morgan`	1.9.0	remote code exec.	2018-3784	9
`merge`	1.1	prototype pollution	2018-16469	15
`mathjs`	3.16.5	command injection	2017-1001002	6
`ns`	all	unsafe serialization	2017-5941	6
`st`	0.2.1	directory traversal	2014-3744	13
`qs`	6.0.0	introspection, poisoning	—	11
`s/js`	0.4.8	runtime evaluation	—	18
`fernet`	0.0.9	timing attack	—	—
`uri-js`	2.1.1	denial of service	—	∞
`libxml`	0.16	unsafe extension	—	∞
`hostr`	2.3.2	read file-system	—	18
`arg.js`	—	expose process args	—	4
`arr.js`	—	inspect array values	—	5
`env.js`	—	environment variables	—	4
`glob.js`	—	access globals	—	4
`mod.js`	—	read module cache	—	6

Take-Away IRIS *can successfully protect against both known and new attacks that fall within its threat model.*

7.3 Expressiveness Evaluation

How expressive are the PICs? In addition to the contracts in Table 3, we develop three more sophisticated contracts to explore the power of PICs.

Invalid Access The first contract not only monitors but also reports a compartment's number of invalid attempts to access global variables. It is simple enough to fit in a dozen lines, even if we account for loading the rules and writing the report to disk:

```
let fs = require("fs"), e = {};
let ad = require("./paths.json");
forevery.global.in(/serial/).do({
  call: (name, path, _) => {
    if (!ad[resolve(path, name)]) {
      e = e[path]? e[path] + 1 : 1;
    } } });
process.on("exit", () => {
  fs.writeFileSync("exceptions.json",
                   "utf-8",
                   JSON.stringify(e)); });
```

A `paths` JSON file maps paths in the global table to permissions. The `forevery` construct is part of IRIS's library (§6) for generating sets of names. The `in` field is a method that takes regular expressions matching module identifiers. If not empty, the `call` hook is called upon access of elements specified in the set. Method `resolve` traverses an object given a

path within that object. Upon program exit, the results are written to disk. Reading specific files and reporting results use the expected language abstractions (available within the PIC compartment).

Allow-Deny We develop two variants of a more comprehensive contract, encoding an allow-deny permission model at the library boundary. The first variant governs individual fields by permission sets containing "on" and "off" permissions. The second variant leverages PICs' Turing-completeness to discover required accesses, generating allow-deny permissions based on the program's runtime behavior: during the training phase, the contract dynamically traces normal accesses, and then later flags any divergences from the observed behavior. Both variants exclude `valueOf`, `toString`, `Promise`, and `prototype` for backward-compatibility reasons (§6). These two contracts amount to 102 LoC and 131 LoC for checking and inference, respectively.

Take-Away *PICs are highly expressive, allowing meaningful end-to-end contracts that use language constructs made available to the PIC compartment.*

7.4 Scalability Evaluation

How does IRIS *perform in larger applications?* To answer this question, we run two experiments on multi-module applications. Our goal is to focus on potential compatibility and interface rewriting issues that challenge conventional MLC (§2.2), even in the full absence of disallowed behaviors (achieved by generating PICs that permit all calls). PICs update invocation metadata to their shared global state, to avoid the V8 engine optimizing them away.

Synthetic Application In the first experiment, we apply BREAKAPP and IRIS on a small synthetic application comprised of two modules, each one importing two additional modules (total: six imports). Modules hold a consecutive integer as their private state. When called, each module increments its integer by six and appends to a global array, which should always be sorted. We run the benchmark for 10K repetitions of 1M calls. BREAKAPP results in at least some elements out of order for all 10K repetitions (**C1**). IRIS always results in a sorted array for all 10K repetitions—that is, PIC execution does not introduce any behavioral changes due to interleaving. We were not able to detect performance differences between the vanilla and IRIS-augmented application (beyond statistical noise).

Real Application In the second experiment, we run IRIS on a wiki application comprised of 130 top-level modules, 1640 total modules, and about 597K lines of code. We use `wrk2` [64] from the same physical host to send HTTP "No-Op" requests. The `wrk2` tool produces a load of constant throughput and calculates latencies at various tail percentiles (*e.g.*, 90%, 99%, 99.9% *etc.*). We configure `wrk2` with 2 threads and 100 connections, at about 1.07k requests per second.

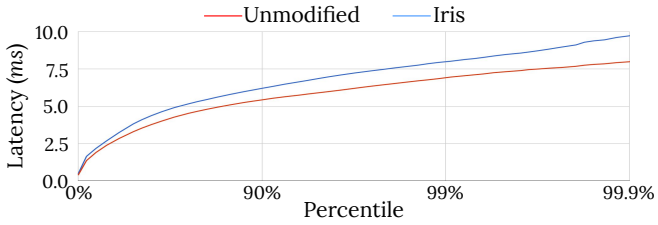

Fig. 11: Performance of IRIS-augmented wiki. High-resolution cumulative latency distribution, highlighting tail latencies at various percentages.

The first result is that applying IRIS did not require any manual effort rewriting interfaces. In fact, we did not need to modify any program code (*i.e.*, apart from providing contracts) to achieve compatibility (**C2**). In terms of performance (**C4**), the unmodified wiki responds with an average latency of 4.2ms; introducing IRIS bumps latency to 4.8ms. Fig. 11 shows that IRIS introduces sub-millisecond overhead for the majority of the requests (90%). Changes in memory consumption, which averages around 100MB, remain under 0.01%.

Unfortunately, we were not able to scale BREAKAPP beyond two compartments. Having a trusted-untrusted pair of compartments is typical of SMLC, but in our case this choice was guided by the fact that IRIS (1) affected stateful libraries, often introducing inconsistencies with respect to their sequential execution (**C1**), or (2) affected the type signatures of synchronous interfaces, requiring manual conversion to asynchronous ones (**C2**). As we did not apply any program modifications, we were left with a small intersection of libraries where BREAKAPP's application did not require such modifications. Response latency was not affected significantly, as the compartment held configuration data, but memory consumption increased by about 30%.

Take-Away IRIS *avoids introducing incompatibilities or interface rewriting, and scales to hundreds of compartments, making large-scale MLC feasible.*

8 Related Work

Compartmentalization Compartmentalization is a hardening technique with a long systems heritage [1, 56, 57]. Much of the work after OpenSSH [55] focused on addressing challenges **C1–4** with compatibility automation [10, 25, 29, 31, 68], concurrency control [26, 71], and performance [6, 12, 73], IRIS leverages language-based techniques to avoid **C1–4**, additionally offering Turing-complete configurability, foregoing protection against native libraries.

Wrapping and Proxying IRIS shares characteristics with prior work on dynamic wrapping [3, 14, 38, 41–44]. This work is different from IRIS as it often (1) offers deeper isolation (including deep attenuation and argument wrapping) but on a narrower set of interfaces [3, 13, 14], (2) does not target unmodified programs, allowing some but not all of the language

constructs [37,43,44,65], and (3) it does not apply load-time name shadowing and thus does not provide multiple virtual library contexts nor top-level isolation [3,5]. While systems exist that could, in principle, offer the desired compartment-like isolation characteristics [3,14,44,65], to the best of our knowledge no previous language-based system has actually done so.

Software Isolation Replacing OS protection mechanisms with language-based ones is not a new idea. Software fault isolation [70] rewrites the object code of modules written in *unsafe* languages to prevent them from writing or jumping to addresses outside their domains. Singularity's software-isolated processes [4] ensure isolation through software verification. Leveraging memory safety, IRIS can be applied in environments with *runtime code evaluation*, for which binary rewriting and software verification might not be an option.

Capability Systems IRIS uses ideas from the capability [33,34,59] and object-capability [45,46] literature. Rather than completely restricting naming, IRIS dynamically shadows names defined outside the compartment's context—a technique isomorphic to a combination of capability revocation followed by attenuation. IRIS' shadowing works with unmodified runtime—different from efforts such as Caja [43] and Joe-E [39] that restrict programming languages to capability-safe (or authority-safe [37]) subsets.

Contracts IRIS' PICs are a special case of contracts—executable partial program specifications—for specifying privilege. Contracts were pioneered by Eiffel's "design-by-contract" methodology [40] and have seen widespread use [23,27,30,53,69]. PICs fit in *latent* contracts [23], purely dynamic checks that are transparent (and orthogonal) to the type system, as opposed to *manifest* contracts [24], in which types record the most recent check that has been applied to each value. IRIS restricts the privilege of code in the contracts themselves, by leveraging its security-monitor transformation infrastructure.

9 Discussion: Other Programming Languages

To better understand how the IRIS approach could be adapted to other programming languages than JavaScript, we discuss several necessary conditions for such an adaption. LMLC makes use of several language features, found primarily in dynamic languages.

First, IRIS re-wires the `import` keyword: if it is a function (*e.g.*, JavaScript, Scheme, Racket), re-wiring amounts to an assignment; if it is a special non-functional keyword, re-wiring needs to be prefixed by a source rewriting pass replacing `import` with a function (upon loading, Fig.4-2).

Second, IRIS 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.

Third, IRIS makes use of weak-reference hash-maps to check for object equality, cycles, and prior wrapping. Weak references 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, the source code transformations require the code to be available, e.g., as a string. IRIS makes use of variable shadowing to hide the original, unmodified versions of values—an almost-universal language feature.[5] In IRIS, all this is achieved conveniently upon loading.

10 Conclusion

Widespread use of module-level compartmentalization is hindered by several challenges, such as the introduction of incompatibilities, manual synchronous-to-asynchronous conversions, coarse-grained privilege control, and significant performance overheads. To address these challenges, this paper presents a new technique termed language-based module-level compartmentalization (LMLC) along with a prototype implementation, IRIS. LMLC effectively replaces OS protection mechanisms with ones provided by the programming language. Rather than relying on the OS to restrict the system-call interface accesses at the compartment boundary, IRIS controls how compartments access functionality by shadowing the names accessed by imported modules. This is enabled by 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 security-specific code into the module before it is loaded. A set of fine-grained executable access predicates termed privilege-interface contracts is responsible for configuring the privilege available to compartments. Experiments with IRIS, our prototype targeting the JavaScript ecosystem, demonstrate significant benefits in terms of compatibility, manual effort, protection granularity, and order-of-magnitude improvements in runtime performance over state-of-the-art MLC.

The presented design shows that it is possible to achieve well-performing compartmentalized systems by carefully replacing operating-system protection mechanisms with language-based ones.

Availability IRIS source code is available on GitHub [blind], obtainable for system-wide installation via Node's `npm`:

```
npm install -g @blind/iris
```

[5] A notable exception is the language CoffeeScript.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Technical Conference*, 1986.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[3] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. Jsand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 1–10, New York, NY, USA, 2012. ACM.

[4] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[5] Devdatta Akhawe, Frank Li, Warren He, Prateek Saxena, and Dawn Song. Data-confined html5 applications. In *European Symposium on Research in Computer Security*, pages 736–754. Springer, 2013.

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[7] Matthias Blume and David McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

[8] Oscar Bolmsten. Malicious package: crossenv and other 36 malicious packages. https://snyk.io/vuln/npm:crossenv:20170802, 2017. Accessed: 2019-03-19.

[9] John Brant, Brian Foote, Ralph E. Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 396–417, Berlin, Heidelberg, 1998. Springer-Verlag.

[10] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[11] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 516–519. IEEE, 2015.

[12] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 65–82, Berkeley, CA, USA, 2018. USENIX Association.

[13] Tom Van Cutsem. Membranes in javascript. https://tvcutsem.github.io/js-membranes, 2012. Accessed: 2020-03-16.

[14] Willem De Groef, Fabio Massacci, and Frank Piessens. Nodesentry: Least-privilege library integration for server-side javascript. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 446–455, New York, NY, USA, 2014. ACM.

[15] Erik DeBill. Module counts. http://modulecounts.com, 2009. Accessed: 2018-11-18.

[16] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Nik Sultana, Bowen Wang, Jingyu Qian, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. A demonstration of the dedos platform for defusing asymmetric ddos attacks in data centers. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pages 71–73, New York, NY, USA, 2017. ACM.

[17] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, 2015. ACM.

[18] Christos Dimoulas. *Foundations for Behavioral Higher-Order Contracts*. PhD thesis, Northeastern University Boston, 2012.

[19] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming*, pages 214–233. Springer, 2012.

[20] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Measuring and preventing supply chain attacks on package managers. *arXiv preprint arXiv:2002.01139*, 2020.

[21] Stéphane Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object Oriented Programming*, 12:39–50, 1999.

[22] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, 2002.

[23] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[24] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 353–364, New York, NY, USA, 2010. ACM.

[25] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1016–1031. ACM, 2015.

[26] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 393–405, New York, NY, USA, 2016. ACM.

[27] Matthias Keil and Peter Thiemann. Treatjs: Higher-order contracts for javascripts. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 28–51, 2015.

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

[29] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[30] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010.

[31] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 51–57, New York, NY, USA, 2017. ACM.

[32] 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.

[33] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM.

[34] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

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

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

[37] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*, pages 125–140. IEEE, 2010.

[38] Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld. Architectures for inlining security monitors in web applications. In *International Symposium on Engineering Secure Software and Systems*, pages 141–160. Springer, 2014.

[39] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *Networked and Distributed Systems Security*, volume 10 of *NDSS'10*, pages 357–374, 2010.

[40] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[41] Leo A Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE, 2010.

[42] James Mickens. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*, pages 261–275. IEEE, 2014.

[43] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, 2008. *Google white paper*, 2009.

[44] Mark S Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in javascript. In *European Symposium on Programming*, pages 1–20. Springer, 2013.

[45] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[46] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, MD, USA, 2006. AAI3245526.

[47] CVE MITRE. Common vulnerability enumeration. 2006. Accessed: 2018-11-18.

[48] CWE MITRE. Common weakness enumeration. 2006. Accessed: 2018-11-18.

[49] Derek G. Murray and Steven Hand. Privilege Separation Made Easy: Trusting Small Libraries Not Big Processes. In *Proceedings of the 1st European Workshop on System Security*, EUROSEC '08, pages 40–46, New York, NY, USA, 2008. ACM.

[50] 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.

[51] npm, Inc. Malicious Package: stream-combine. https://www.npmjs.com/advisories/765, 2019. Accessed: 2019-01-25.

[52] npm, Inc. Malicious Package: stream-combine. https://www.npmjs.com/advisories/774, 2019. Accessed: 2019-01-25.

[53] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, May 1972.

[54] Niels Provos. Improving host security with system call policies. In *Usenix Security*, volume 3, page 19, 2003.

[55] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[56] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[57] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[58] Thomas Hunter II (Intrinsic Security). Compromised npm package: event-stream. https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502, 2018. Accessed: 2019-03-19.

[59] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. *EROS: a fast capability system*, volume 33. ACM, 1999.

[60] Howard Shrobe, Robert Laddaga, Bob Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, Tim Hollebeek, and Alexander Egyed. Awdrat: a cognitive middleware system for information survivability. *AI Magazine*, 28(3):73–73, 2007.

[61] Snyk. Find, fix and monitor for known vulnerabilities in node.js and ruby packages, 2016.

[62] 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.

[63] Liran Tal. The state of open source security—2019. https://snyk.io/opensourcesecurity-2019/, 2019. Accessed: 2019-03-19.

[64] Gil Tene. wrk2. 2015. Accessed: 2018-11-18.

[65] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. Javascript in javascript (js. js): sandboxing third-party scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 95–100, 2012.

[66] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. Towards automatic compartmentalization of c programs on capability machines. In *Workshop on Foundations of Computer Security 2017*, FCS'17, pages 1–14, 2017.

[67] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Towards fine-grained, automated application compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 43–50, New York, NY, USA, 2017. ACM.

[68] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Networked and Distributed Systems Security*, NDSS'18, 2018.

[69] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

[70] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[71] Jun Wang, Xi Xiong, and Peng Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 361–373, Berkeley, CA, USA, 2015. USENIX Association.

[72] Robert NM Watson. Exploiting concurrency vulnerabilities in system call wrappers. *WOOT*, 7:1–8, 2007.

[73] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[74] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[75] 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.

A The Node.js Module System

The following subsections outline how module systems handle modules at runtime, by exemplifying the internals of Node.js. It sketches the Node.js runtime (§A.1), the use of the module system (§A.2), and the internals of how module loading works (§A.3).

A.1 JavaScript Implementation

Node.js combines a number of components to provide a high-performance JavaScript runtime decoupled from the web browser: (i) the V8 engine, Google's performance-oriented JavaScript implementation used in Chrome; (ii) libuv, a cross-platform library for event-driven, asynchronous I/O; (iii) a number of utility libraries (*e.g.*, a C library for HTTP parsing, zlib for compression, OpenSSL for SSL and TLS); and (iv) a JavaScript standard library (*e.g.*, `module` for loading modules, `url` for URL processing).

V8 compiles and executes JavaScript source code. It provides mechanisms for creating objects, calling functions, allocating and collecting memory, and isolating code within a single runtime.

Libuv implements the I/O subsystem (*e.g.*, file-system, network) with an emphasis on providing abstractions that (i) are identical between different platforms, and (ii) follow an asynchronous, event-driven style. Its core functionality is comprised of the event loop, its worker threads, and callback-based notifications of I/O and other events (*e.g.*, signals). Moreover, it provides a unified, platform independent API by wrapping such platform-specific APIs for system-level functionality (*e.g.*, POSIX timers, sockets, file operations, signals, system events, execution of children processes *etc.*) with platform-independent, asynchronous equivalents. It gathers events from the operating system (*e.g.*, epoll, kqueue, IOCP, event ports) and invokes callbacks defined by the users specifically as handlers for these events.

A.2 Interface of the Module System

Node.js' module system is implemented entirely in JavaScript. It exposes `require`, a global-looking function for importing modules. The following example demonstrates the use of `require`:

```
1  // --------------- [./main.js] -------------
2  // importing point
3  let Point = require("./point.js");
4  Point.create(1, 1).print(); // => [1, 1]
5
6  // --------------- [./point.js] ------------
7  let Point = function Point (x, y) {
8      this.x = x; this.y = y;
9  };
10 Point.prototype.print = function print () {
11   console.log([this.x, this.y]);
12 };
13 module.exports = {
14     create:  function create (x, y) {
15         return new Point(x, y);
16     }
17 };
```

In the example above, the main module (`main.js`) imports the `point.js` module using the `require` statement (line 3) Functionality from the exporting module (`point.js`) that is expected to become available to the importing module (`main.js`) is assigned to a special `module.exports` object (line 13); the rest is module-private functionality. Files and modules are in one-to-one correspondence (each file is treated

as a separate module). Method `require` is synchronous (*i.e.*, blocking): it will block execution until the module specified is loaded. The module system is implemented in the `module` built-in module (§A.3), which locates, wraps, compiles, and executes the specified file.

IRIS hooks into `module` and alters the return value of the `require` call that imports `point.js` (line 3).

A.3 Implementation of the Module System

At a very high level, loading a fresh module with `require("foo");` corresponds to the following five stages:

1. Resolution: identify the file to which the module specified corresponds, and locate it in the filesystem.
2. Loading: depending on the file type, use the corresponding loader (*e.g.*, V8 compiler for `js`, `JSON.parse` for `json` *etc.*).
3. Wrapping: wrap the module so that module-globals get encapsulated and Node.js globals (*e.g.*, `require`) get resolved.
4. Evaluation: evaluate the wrapped module in the current context, so that global names and top-level objects get resolved correctly.
5. Caching: add the module to a handful of module-related cache structures, for purposes of consistency and performance.

IRIS interposes on all of these steps to facilitate transformations. Wrapping (3) and evaluation (4) are particularly interesting, because they allow IRIS to interpose at the module boundary during runtime. Before a module's code is evaluated, the Node.js module loader wraps the module so that (i) it keeps top-level variables (defined with `var`, `const` or `let`) scoped to the module rather than the global object; and (ii) it provides some global-looking variables that are actually specific to the module, such as the `module` and `exports` objects that the implementor can use to export values from the module and convenience variables—such as `__filename` and `__dirname` containing the module's absolute filename and directory path, respectively. True globals remaining are (i) the global objects as defined by the EcmaScript standard (*e.g.*, `Object`, `Function`, `Math`); and (ii) Node.js-specific globals (*e.g.*, `console`, `process`, `timer`). These globals require further interposition.

```
1 // Node.js will wrap a module with a function,
2 // so as to bring certain names into scope
3 // before compiling/evaluating code.
4 let wrapped = "function (" +
5         "exports, require, module, " +
6         "__filename, __dirname, CTX) {" +
7     "let Math = CTX.Math" +
8     "let console = CTX.console" +
9     //...[more definitions]
10    moduleSource +
11  "});"
```

IRIS hooks into the wrapper function (the last variable in the function definition, `CTX`). This trivial source-to-source transformation re-defines global variables as module-locals and initializes them with IRIS-augmented values. For example, `console` in the context of the module will be an IRIS-created object that allows IRIS to interpose on it. Evaluation of the module passes an additional value to this function, which is the modified context. As a result, any changes to the top-level objects and any global variables are accessible from the within the module.

```
1 //Input: module ID e.g., absolute filepath
2 let load = function (ID) {
3   if (cache[ID]) {
4      return cache[ID];
5   }
6   let m = {
7     exports: {}, id: ID, dir: path.resolve(ID)
8   };
9   let cm = v8.compile(wrapped);
10  let ii = iris.getImplicit(ID);
11  let c = iris.freshContext(ii);
12  cm(m.exports, this.require, m, ID, m.dir, c);
13  let ei = iris.getExplicit(ID);
14  m.exports = iris.wrap(m.exports, [ei] );
15  cache[ID] = m;
16  return m.exports;
17 }
```

The `load` method in the `Module` module combines evaluation (line 10) and caching (line 14) of the wrapped module. After evaluation, invoking the compiled function generates the value that is assigned to `module.exports` from within the module (line 12). IRIS passes a freshly constructed context at that invocation, modified according to the implicit segment of the PIC corresponding to the module being loaded. Before returning the value of `module.exports`, IRIS transforms it according to the explicit segment of the PIC corresponding to the module being loaded. Finally, the results of the entire process are placed into the module cache for later use.