BreakApp: Automated, Flexible Application Compartmentalization

Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, Jonathan M. Smith
University of Pennsylvania
{nvas, karel, nroess, ndd, andre, jms}@seas.upenn.edu

***Abstract*—Developers of large-scale software systems may use third-party modules to reduce costs and accelerate release cycles, at some risk to safety and security. BREAKAPP exploits module boundaries to automate compartmentalization of systems and enforce security policies, enhancing reliability and security. BREAKAPP transparently spawns modules in protected compartments while preserving their original behavior. Optional high-level policies decouple security assumptions made during development from requirements imposed for module composition and use. These policies allow fine-tuning trade-offs such as security and performance based on changing threat models or load patterns. Evaluation of BREAKAPP with a prototype implementation for JavaScript demonstrates feasibility by enabling simplified security hardening of existing systems with low performance overhead.**

I. INTRODUCTION

Software development is changing in scale, process, and basis for trust. Early open-source software such as the Linux kernel or Apache had many people focused on the quality and security of a single codebase. Yet even such cohesive efforts failed to prevent a slate of vulnerabilities [7], [9].

Current software makes extensive use of third-party modules created by different authors and accessed via language-specific package repositories. For example, JavaScript's Node Package Manager [46] hosts more than half a million packages from over 100K authors and serves hundreds of millions of package downloads *per day*. Such public repositories provide no guarantees on modules beyond availability; anyone can create an account and share packages.

A sample of large-scale applications (Section IX) shows that foreign code accounts for up to 99.9% of that released to clients, and thus most code is neither written nor reviewed by its nominal developers. In practice, glue code stitches together many specialized modules comprising the application into a system with deep, intricate interdependencies. As we show, several hundred third-party dependencies occur in an average application due to recursive imports. This gives rise to security vulnerabilities, as these modules execute with no isolation or privilege separation beyond what type safety provides.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23131
www.ndss-symposium.org

Further problems increase these risks. With popular modules averaging tens of thousands of lines of code, understanding the internals of a complex package and verifying that it will not behave in unintended ways [15], [33] are both extremely difficult tasks. The popularity of certain packages—depended-upon by thousands of other packages—allows vulnerabilities deep in the dependency graph to cause widespread difficulties [30], [23], [60]. Discovered vulnerabilities are becoming harder to eradicate, since some updates are fetched automatically [43], and module unpublishing is becoming a multi-step process in order to avoid breaking dependency chains [59].

Software supply chain attacks are becoming an important concern. Instead of merely reacting to announced vulnerabilities [28], [8], [51] or avoiding composition altogether due to security concerns, *we propose leveraging the trend towards more and smaller modules to enhance, or retrofit, application security*. The core idea is to exploit programming language properties (*e.g.*, abstraction, encapsulation, trust boundaries) to automatically transform a program at the module boundaries and offload enforcement to the operating or runtime system (*e.g.*, address space isolation, LXC/namespaces, sandboxing).

BREAKAPP is a drop-in replacement for a language runtimes module system that pioneers the use of module boundaries as a guide to placing code into protected compartments. BREAKAPP is centered around a *parametrizable* transformation technique that spawns modules in their own compartments during runtime. Automated transformations (*e.g.*, function calls to remote procedure invocations, garbage collection propagation) hide compartment boundaries, providing the benefits of compartmentalization with low developer effort.

Optional runtime policy expressions fix the aforementioned parameters, effectively decoupling assumptions made during module development from requirements present during module composition. Certain powerful linguistic features, such as introspection and global variables (Section III-A), pose high risks for inter-module attacks. Allowing developers to disable them when *their* side of the code does not use them eliminates classes of attacks. Since the same module can be used by several different applications, each with its own assumptions and sensitivities, it is important to let the application developer choose which module behaviors to disallow based on *their* side of the code instead of whether vulnerabilities for the modules in use have already been discovered. Moreover, the aforementioned transformations for creating compartments and maintaining the illusion of a single runtime open a rich space of security and performance trade-offs. Thus policies can also improve BREAKAPP's performance by allowing programmers to customize the provided functionality on a per-import basis.

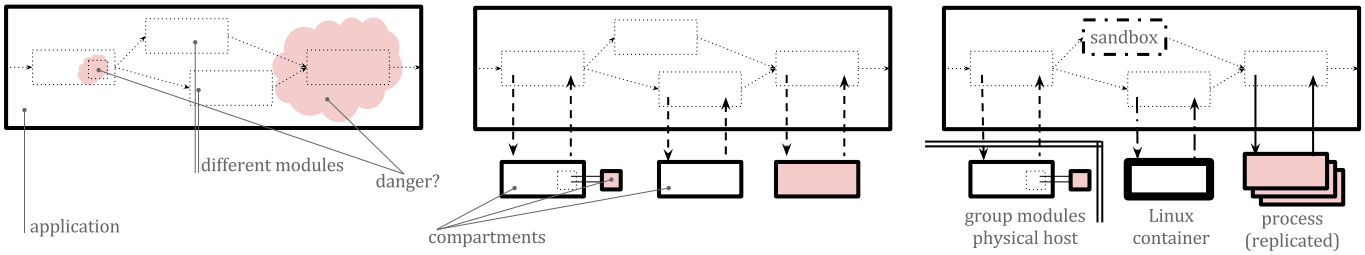

Fig. 1. A simplified server application with multiple third-party modules of varying trust.

BREAKAPP does not require any annotations, does not require any tracing or inference (pre-)runs, and does not require manual rewriting of source code. Policy expressions are backward-compatible with existing codebases and forward-compatible with unmodified module systems. The system lowers potential barriers to widespread adoption and makes incremental security retrofit in existing systems possible.

We demonstrate a prototype of our system, `breakapp`, targeting JavaScript. We leverage JavaScript's flourishing package ecosystem to show that it mitigates several classes of discovered vulnerabilities, as well as wider classes of hypothetical vulnerabilities. We show that good parameter choices can give acceptable performance results, hitting a sweet spot between security and performance.

Our contributions include:

- identifying an opportunity in today's applications; the use of many third-party modules, although risky, offers clear boundaries of trust (Section III).
- formulating a parametrizable technique for automatically transforming modules to standalone compartments, allowing users to compartmentalize applications at the boundaries of untrusted modules (Sections IV, VI).
- proposing a concrete set of policies for configuring the aforementioned parameters, effectively allowing users to fine-tune the security, compatibility, and performance trade-offs (Section V) during runtime.
- building an open source, prototype implementation of our technique (Section VIII), and evaluating applicability, security, and performance, leading to configurations that work well in practice (Section IX).

II. OVERVIEW

While our concerns with third-party code are language-independent, the problem and proposed solution are developed here for an interpreted language where source code is available. We sketch additional challenges for compiled languages in Section VII.

A. A Blogging Platform

To highlight typical module usage in modern applications, we consider Ghost, an open source blogging platform. Ghost imports 62 top-level packages and makes use of 981 packages in total. The snippet below presents a simplified version of the core functionality behind such a blogging platform's search capability:

```
1 var dbc = require("./dbc.json");
2 var ejs = require("ejs");
3 function search(req, res) {
4   var f = db.getFiles(dbc);
5   var m = require("minimatch");
6   var r = m.match(f, req.query);
7   res.body = ejs.render(template, r);
8 }
```

Function `search` takes a request and a response object. It populates results in the response object based on query data from the request and the result-page template from the developer. Each `require` statement imports a module into the current scope. More specifically, it returns the value assigned to the module's `module.exports` variable.

Fig. 1-a shows the simplified application running. Boxes correspond to the context of different modules, with the outer box corresponding to the top-level context. All these *logically unrelated* packages execute within the same address space; a problem in any one of the packages exposes other packages, too. But what can go wrong when we are talking about a high-level, memory-safe, managed programming language?

Problem 1 As a simple example, suppose that Ghost generates HTML from templates using `ejs` (line 2). Since this package is susceptible to remote code execution (Table I), a malicious version of this module or a user using the service could try to get access to the database credentials (`dbc`) by a number of different execution paths: (i) attempt to read the global, singleton `dbc` object by taking advantage of JavaScript's default-is-global variable resolution mechanism, the complex `this` semantics, or by reaching to the caller's environment if strict mode is not being used; (ii) access the `dbc` object by dynamically patching the top-level object and interposing on its access; (iii) access the loaded config module by traversing the cache of loaded modules; or (iv) directly import the `dbc.json` config file.

Problem 2 Suppose Ghost provides search functionality using the `minimatch` module, which converts "glob" expressions to regular expressions (line 5). Even if we assume that `minimatch` itself is benign, our application is still vulnerable. Because it is supplied user-generated strings, a malicious user can launch a RegEx DoS attack by providing pathological regular expressions. Since most JavaScript implementations follow an event-driven, cooperative concurrency model, a

problematic search query will cause the application to freeze until the pathological request completes.

B. Strategy: Disabling Features

Language features illustrated in Section II-A above and detailed in Section III-A below are good to have some of the time, and may be vital in certain cases. For example, users should be able to introspect and rewrite state dynamically, share global variables, compile code, access unsafe code, and traverse the module cache. However, with the prevalence and simplicity of third-party code there are cases when users need to *selectively* disable some of these features. *Within* their dedicated compartments, modules should still have unrestricted access to all these features, but some of them should stop exactly at the compartment boundary. It is the module client who should decide which features are allowed to cross boundaries and which ones cannot. This insight is the driving force behind the design of automated compartmentalization.

The first challenge is giving users the ability to build boundaries within an application as easily as importing third-party packages in their application. This is solved by providing a technique, discussed in Section IV and illustrated in Fig. 1-b, to transparently spawn modules in their own fresh compartments. Isolated modules can only communicate through a tight interface (the module's API) and cannot access state from other modules without using the API. In the earlier example of a simplified blogging platform (Section II-A), the system can spawn `ejs` in a compartment separate from `dbc.json` and `minimatch` in multiple compartment replicas.

The second challenge is giving users the ability to select the behaviors they need to disable between boundaries and when. This is solved through a policy scheme, discussed in Section V, that allows users to parametrize several aspects of the compartment types for a particular runtime instance (Fig. 1-c). Even if two applications use exactly the same third-party modules, their safety assumptions, use of these modules, the intended deployment environment, and the sensitivity of the non-third-party code justifies disabling very different features — irrespective of whether and what vulnerabilities have been discovered in these modules. For example, a development machine could place all third party code (*i.e.*, *both* `ejs` and `minimatch`) in a single compartment with restricted access to the developer's file system.

The third challenge is having the compartmentalized application maintain the semantics of the original application executing on a single runtime. This is solved using further transformations, discussed in Section VI. Such transformations include converting pointers to distributed references, propagating changes in copied primitive values, and reflecting garbage collection events. For example, when a developer decides to manually unload a module using the interactive interpreter, this should lead to the destruction of the respective compartment and reclamation of its resources.

III. Background, Threats, and Opportunities

This section discusses more problems related to third-party modules and outlines which threats fall within our model.

A. More Problems

The two examples presented in Section II-A only scratch the surface of the risks posed by problematic modules. In this section we outline further potential problems. These problems (except (c)) are not specific to JavaScript; substantially similar problems affect languages such as Java, Go, Objective-C, Ruby, and Python. We give each problem a circled letter to allow us to refer back to them when explaining how our system can mitigate each problem (*e.g.*, Table III in Section V-B).

Developer Intentions A common source of problems has to do with *development patterns or mistakes*. A common pattern with unintended consequences is global variables ⓐ; for example, having a global, singleton object for an application-wide database or logger configuration [16], [18]. Mistakes include accidentally exposing state at the wrong level ⓑ or making a typo [55] while importing a package (*e.g.*, is it "coffeescript" or "coffee-script"?).

Runtime Capabilities Another set of problems has to do with powerful *reflection and introspection capabilities*, available in many programming languages. These allow any part of the program, including a buggy or malicious module, to examine ⓒ and alter ⓓ the program's own structure and behavior during runtime. Implementations often allow inspecting the call stack for debugging purposes, enabling code to read data from the calling context ⓔ. Runtime code evaluation with `eval` or `exec` in many dynamic languages allows malicious code to hide its intentions ⓕ.

Language and Runtime Environment Further problems can arise from *the design of the language*. In the case of JavaScript, for instance, common problems include: default-is-global, where resolution of a name not in scope continues to outer contexts until it reaches the global context (where it might hit something valuable) ⓖ; prototype poisoning, where code at inner contexts can affect objects higher in the prototype chain (effectively mutating all children) ⓗ; a wide range of mutability attacks, where code can edit properties or even rewrite code before calling it ⓘ, meaning there is no guarantee that your code will run as expected; self-reference (*i.e.*, `this`) semantics that might resolve differently depending on how the object is being called ⓙ. Other problems are *implementation-specific*; for example, several high-performance implementations employ cooperative concurrency, choking at seemingly innocuous calls that block the event loop ⓚ [1].

The Module System A largely undiscovered set of problems has to do with *the inherent implementation of the module system*. Module systems do not have the notion of authority: everything is accessible at any point during the execution of the

TABLE I. Eight major vulnerability classes and specific instances of packages available on npm; "++" indicates that many more packages with similar problems exist.

Problem	Example Package
Directory Traversal ⓛ	hostr, bitty, restafary, ++
Denial of Service ⓚ ⓞ	**ejs**, node-uuid, **minimatch**, ++
Remote Code Execution ⓒ ⓓ	**ejs**, pouchdb, reduce-calc, ++
Timing Attack ⓟ	fernet, cookie-signature, ++
Uninitialized Mem. Exposure ⓔ	mongoose, bl, request, ws, ++
Command Injection ⓕ	git-ls-remote, shell-quote, ++
Native Code Vulnerabilities ⓠ	libxmljs, libyaml
Sensitive Info. Exposure ⓝ	airbrake

program. A malicious module can use built-in modules to access system resources with the same authority as the rest of the application ⓛ, with code such as `fs = require("fs"); fs.readFile("/etc/passwd")`. Worse, module systems tend to cache loaded modules to avoid overheads from loading and to ensure consistency of any state within the module. As a result, a malicious module can enjoy direct, unrestricted access to the latest instance of any module that is already loaded (*e.g.*, `require.cache`) ⓜ. In many cases, it can even read or write built-in functionality (*e.g.*, overwrite built-in `fs` handlers) ⓜ+ⓘ.

The Broader Environment Some problems are more general and have to do with *the wider system* on which an application is running. For instance, modules have direct access to the calling environment with `process.env` and `process.args` ⓝ. Malicious code within the module or malicious input from users aware of its use can accidentally or intentionally exhaust resources ⓞ. Modules can also leak timing information about their state, computation, or underlying resources from observable changes of how long it takes to execute a request ⓟ.

Native Modules Finally, modules can interface with code written in other languages. Reuse of existing libraries via foreign functions is a compelling proposition, but use of unsafe code nullifies the safety guarantees of a high-level, memory-managed programming language ⓠ.

B. Threat Model

The general model of threats arising from the use of modules is quite broad. In practice, however, users are expected to shield applications against only a specific subset of these threats.

Source of Attacks In terms of attack origin, we care about three broad types: (i) a malicious module directly attempting an unintended action (*e.g.*, cause a DoS attack by looping infinitely); (ii) a malicious module indirectly coercing a different module in the dependency graph with a known vulnerability into performing an unintended action (*e.g.*, cause DoS by carefully using another module's API); (iii) a user feeding a problematic module input that triggers an unintended action (*e.g.*, cause a DoS by submitting problematic search queries).

Attacks We want users to be able to protect against code that attempts to violate the confidentiality (*e.g.*, read global state, load other modules, exfiltrate data) and integrity (*e.g.*, write global state or tamper with the module cache) of application data and code. Moreover, the code can attempt to read or write the broader environment within which the application is executing, including environment variables, hardware counters, the file system, or network. We want to mitigate these attack vectors by allowing users to disable access to specific variables, specific modules, or system-level capabilities, such as file system or networking primitives.

We additionally seek to weaken attacks on availability. Pathological inputs from attackers can disrupt otherwise benign modules within an application (*e.g.*, RegEx matching [10] or JSON parsing [47]). Potential mitigations range from simple reporting, to back-pressuring malicious input, to decreasing resources of malicious compartments, to shutting down offending compartments.

We also want to make it possible for users to shield time-sensitive modules from timing attacks. In particular, we want to allow users to set specific minimum response times for cross-boundary calls.

Assumptions We assume that the core language runtime and built-in libraries such as `fs` and `net` can be trusted. As we show in Section IX, our technique allows spawning built-in modules in their own dedicated compartments. However, the system requires a minimum of trusted functionality from the underlying system, such as the ability to locate and load the right program files required by a module. This can be achieved by including a trusted (say, formally verified) version of a mini standard library (*e.g.*, enough to locate files, load modules *etc.*). We do not explore these options in this work.

BREAKAPP can be run either as a third-party module loader on a per application basis or as a system-wide module loader replacement. In the former case, we assume that users load BREAKAPP *before* any other module; otherwise, a malicious module could dynamically rewrite BREAKAPP's code. Using a defense-in-depth approach, BREAKAPP checks whether other modules attempt to rewrite any of its core structure using both static checks (the moment BREAKAPP loads other modules, it parses their source code) and dynamic interposition hooks on its internal objects (Section VI-E). Moreover, most of its core structure is immutable: hidden object properties are set to non-enumerable, non-writable, and non-configurable modes; and policies are by-default frozen after the initial configuration.

Limitations Attacks targeting package managers are related to, but distinct from, those we protect against. Most package managers implement pre-install, post-install, testing, and other scripts that are package-specific. Since these scripts are Turing-complete programs similar to full-fledged modules, they can be used to launch attacks to the system before, during, or after package installation similar to the ones described earlier (*e.g.*, read environment variables, denial-of-service attacks *etc.*). However, these are beyond the scope of this work and are better addressed with other methods [6].

IV. TRANSFORMATIONS: MODULE DECOMPOSITION

This section discusses automation related to spawning modules in their own dedicated compartments.

At its core, BREAKAPP changes the implementation of all module import statements to (i) spawn a new compartment for each previously unseen module, (ii) modify the return value so that use of module's members (property accesses and function calls) will invoke RPC proxies to the newly-spawned compartment, and (iii) redirect module-specific side-effects, such as console output or exceptions, to the importing compartment. BREAKAPP also monitors the health and status of all compartments.

A. Compartment Setup and DAG Transformations

Whenever the program executes an import statement, control jumps to BREAKAPP. Consulting the policies (Section V) associated with this import statement, it chooses whether it should spawn a new compartment. If the policy dictates that it should, it creates a new *child* compartment for the imported

```
transform (e : DAG) (toRPC: Fun -> Fun) : DAG :=
match e with
| Obj ((k, v) :: xs) => Obj ((k, transform v) :: mt xs)
| Arr x::xs => Arr ((transform x) :: mt xs)
| Fun f => toRPC(e)
| _ => interpose(copy(e))
end
where mt = map transform
```

Fig. 2. The core transformation; example result in Fig. 3

module and sets up a new communication channel between the two. It replaces core functionality on the child compartment, such as console printing, in order to propagate certain side-effects to the parent compartment.

Within the child, BREAKAPP copies and transforms the return value for the raw imported module before sending it to the parent compartment. The general case of such a value is a directed acyclic graph (DAG). The system walks the DAG and transforms its component values so that function and method calls propagate to the compartment.

The exact transformation is parametrizable on several aspects related to policies, but it can be summarized as follows:

- *primitive* values are copied unmodified and wrapped with an interposition mechanism that records changes.
- *function* values are replaced with an RPC stub that, when called, will serialize arguments, send them to the current compartment, and deserialize the return values.
- *objects* are recursively copied and transformed, with their getter and setter functions replaced with RPC stubs similar to function values.

If the specified module throws an exception while being loaded, the exception is caught by BREAKAPP running on the child, serialized, and re-thrown by the parent compartment.

If the specified module is already loaded and the policy associated with this import statement allows module reuse, BREAKAPP simply retrieves the channel pointer and returns the previously-transformed DAG copy. Fig. 2 summarizes the transformation algorithm. We discuss an example transformation (Fig. 3) at the end of Section VI.

B. Function Calls as RPCs

BREAKAPP mediates between the parent and child compartments. Synchronous calls yield to the module scheduler, which serializes arguments, sends them through the channel to the child, and blocks for a response. The child-side wrapper deserializes arguments, calls the required method, and sends results back through the channel. For asynchronous function calls, the parent module wrapper registers an event that invokes the provided continuation (with the available results) when a result is made available on the channel.

In cases when something does not go as expected in the child's execution, its code will throw an exception which BREAKAPP serializes and returns to the caller compartment. BREAKAPP on the parent compartment will inspect the exception and, if it is related to any violations (*i.e.*, it is not an exception coming from BREAKAPP itself), it will re-throw it. BREAKAPP-specific exceptions are handled specially, depending on the violation.

V. POLICIES: TUNING TRADE-OFFS

This section discusses policies and how they automate control over tradeoffs among security, compatibility, and performance.

A. Expressing Policies

Policies (Table II) can be expressed both at the level of the whole application and the level of each module. In both cases they are optional. BreakApp's default policy is overriden by application-wide policies, which are in turn overriden by per-module policies.

Application-wide policies generally describe coarse guidelines on how to decompose the application. Typical coarse guidelines include the maximum number of compartments (`LEVEL`), action to take in case of violations (`ON_FAIL`), compartment type (`BOX`), and application-wide global variables (`CONTEXT`). They can be expressed at the point of BREAKAPP's initialization:

```
require("breakapp")({box: require.boxes.SBX});
```

The line above specifies that all modules should be loaded in their own, fresh, software-isolated sandboxes (`SBX`). It creates a new runtime context with fresh built-ins and top-level objects for each module.

Module-specific policies give developers fine-grained control over decomposition, allowing them to capture intuition about the properties (regarding security or performance) of the modules they use. Per-module policies are expressed at the module's import statement:

```
require("minimatch", {box: require.boxes.PROC});
```

The line above specifies that the `minimatch` module should be loaded in its own, fresh process. It creates a new address space, and lets the operating system provide support for isolation, scheduling, and interprocess communication. If `minimatch` is a module written in C, it cannot even forge a pointer to poke into the main application's address space. However, it may take a bit longer to load and communicate than the rest of the sandbox-based compartments.

The combination of the two policies above should now be clear: load each module in its own software-isolated sandboxes but load `minimatch` in a new process. If `minimatch` is already loaded in its own compartment, BREAKAPP will spawn a new instance of `minimatch` in its own process.

Notable characteristics of policy expressions include:

Generation: In all cases the policy object can be generated programatically during runtime (*e.g.*, from command line arguments, from the environment, or through a pre-processing stage). This gives programmers considerable flexibility, and allows tools built on top of BREAKAPP to generate policies

TABLE II. EXAMPLES OF INTERESTING POLICIES.

Policy	Example Options	Explanation	Section
BOX	`SBX, PROC, LXC`	Compartment type	V-B
IPC	`TCP, UDS, FIFO`	Communication type	V-B
CONTEXT	`{global: global}`	Share pointers with parent	V-B
LEVEL	`0, 1, ..`	Depth at which to decompose	V-C
GROUP	`subtree.json`	Group dependency subtrees	V-C
TRUST	`["fs", "http"]`	Whitelist trusted modules	V-C
DOUBT	`["ejs"]`	Blacklist untrusted modules	V-C
INSTANCES	`FUSE, PART`	Fresh compartment per import	V-D
REPLICAS	`true, 23`	Multiple replicas (round-robin)	V-D
ONFAIL	`(e) => {..}`	Action upon failure (function)	V-E
COMPOSE	`OURS, THEIRS`	Priority in policy conflicts	V-F

TABLE III. DIFFERENT COMPARTMENT TYPES AND PROBLEMS THEY MITIGATE: VANILLA MODULE SYSTEM (NONE), SANDBOXES (SBX), PROCESSES (PROC), AND CONTAINERS (LXC)

	NONE	SBX	PROC	LXC	Notes
ⓐ ⓑ	✘	✔	✔	✔	globals, state
ⓒ ⓓ	✘	✔	✔	✔	introspection
ⓔ	✘	—	✔	✔	stack inspection
ⓕ	✘	—	✔	✔	evaluation
ⓖ ⓗ ⓘ ⓙ	✘	—	✔	✔	context
ⓛ	✘	✘	✘	✔	fs, net (leaks)
ⓜ	✘	✘	✔	✔	module cache
ⓝ	✘	✘	✔	✔	process args
ⓝ	✘	✘	✘	✔	process env
ⓚ ⓞ	✘	✘	✔	✔	denial of service
ⓟ	✘	✘	✘	✔	side-channels
ⓠ	✘	✘	✔	✔	unsafe extensions

dynamically in response to changing load patterns or evolving threat models.

Compatibility: Per module policy expressions are *fully* compatible with existing codebases. Expressing policies is *backward*-compatible with systems that do *not* provide a BREAKAPP-enabled module system; due to variadic arguments, the policy argument is ignored by the built-in `require` function. Not specifying policies (*i.e.*, all of the code out there today) is *forward*-compatible with systems that *do* provide a BREAKAPP-enabled module system: as explained earlier, BREAKAPP will use the application-wide default configuration.

Extensibility: Policies are extensible. BREAKAPP allows users to override most of the functionality during initialization (*i.e.*, the application-wide policies described earlier) by passing in functions. The sets of policies described here are simply default extensions bundled together with the system. If users need to provide further functionality (*e.g.*, use a different type of compartment or take different actions upon failure), they can hook up their own implementations.

B. Isolation Primitives

Different compartment types provide different guarantees in terms of isolation, but also affect performance directly. Table III shows how different isolation primitives mitigate different types of problems described in Section III-A:

- Sandbox Isolation (`SBX`): This creates a new software-isolated context within the same runtime. Built-in utility functions (*e.g.*, `Math.pow`) and top-level objects (*e.g.*, `Function.call`) are fresh, and global variables not explicitly white-listed are not shared. The sandbox shares the same heap and event queue with the rest of the application.
- Address Space Isolation (`PROC`): This creates a new runtime instance as a new process, with its own address space, stream and IPC handles. The system leverages the OS kernel to synchronize communication and set scheduling priorities between compartments.
- Container Isolation (`LXC`): This creates a new runtime instance within a fresh container instance. Containers can restrict process-trees from accessing arbitrary parts of the filesystem. They can also restrict access to the network and set resource restrictions to the use of CPU and memory.

Heavier compartment types and hence more expensive performance costs are positively correlated with better isolation. However, after fixing the compartment type, there is room for further tuning performance and isolation independently of each other. Isolation guarantees can be fine-tuned without affecting performance by declaring which state compartments are allowed to share (`CONTEXT`). For instance, users can share some of the global variables, some of the built-ins, and choose to allow access to the module cache. Performance costs can be fine-tuned without affecting isolation by choosing one of the available communication channel types (`IPC`). For instance, in the case of process-level isolation, TCP streams provide better throughput, but Unix Domain Sockets and Unix FIFO Pipes offer lower latencies (Table VIII).

C. Decomposition Granularity

Decomposition granularity affects how many modules to launch into separate compartments and is directly related to the number of compartments created. This, in turn, is positively correlated with finer-grained security, since there are fewer components to which a piece of code has unrestricted access. The increase in security generally assumes that, all other things equal (*e.g.*, programming language, code paths *etc.*), there is a correlation between lines of code and exploitable bugs [31]. However, a larger number of compartments can affect performance negatively by increasing startup times and communication costs.

There are many ways of guiding the BREAKAPP compartmentalization scope. At a coarse granularity of specification, users have two knobs: vertically, define the level (`LEVEL`) at which to decompose (*e.g.*, only top-level, every level, only last level *etc.*); and horizontally, define the granularity (`GROUP`) of dependency subtrees (*e.g.*, package-level, file-level, *etc.*). At a fine granularity of specification, they can blacklist components that should always launch in a new compartment (`DOUBT`) and whitelist compartments that are trusted and should always stay with the parent compartment (`TRUST`). BREAKAPP already uses module whitelisting to avoid spawning built-in modules and its own trusted dependencies.

D. Instantiation and Replication

Identical-looking import statements might get resolved into different absolute filenames depending on where they are called in the dependency chain. By default, BREAKAPP takes this into account and spawns new compartments only when the vanilla module system would actually import a module. However, users can request BREAKAPP to further *replicate*

a module to address DoS concerns (REPLICAS). Replication requires user insight because modules that encapsulate state have the potential to introduce state inconsistencies. When used, the number of replicas can be declared statically upon startup or inferred dynamically in response to changes in the load and module response rate. Users can also select a scheduling policy (SCHED) from an existing set (*e.g.*, round-robin) or can define and pass a custom one.

E. Other Policies

When a violation is detected, BREAKAPP can select between several actions, depending on the type of the exception (ON_FAIL). Among other things, it can log the violation, email an administrator, kill or restart the compartment, or launch a new replica. Other policies include scanning the module's source code to pro-actively spawn compartments in parallel before they are requested (PRELOAD), encrypting communication between compartments (ENCRYPT), setting minimum response times (TIMER), whitelisting environment variables (ENV), and soft-reloading modules without restarting the compartments (RELOAD).

F. Conflict Resolution

The introduction of BREAKAPP to a package ecosystem will inevitably lead to conflicts of policies. First, third-party packages will start importing other packages using what they think are the right policies. Then, applications importing these packages might choose to use different policies. There is no single way for resolving these conflicts: in some cases the library developer knows better, but in others the top-level application developer knows more about the intended audience—until, of course, their application is used as another application's library.

To solve this, BREAKAPP comes with a number of conflict resolution options (*e.g.*, accept-ours, accept-theirs, accept-most-restrictive, accept-most-permissive). All policies can "lock" at top level, trumping any other policy expression found in third-party modules.

There is no conflict between different versions of BREAKAPP, since there is only one version running: the one starting with the application at top-level. Even if other modules import BREAKAPP later, the BREAKAPP instance that is already loaded will bypass these imports as no-ops (*i.e.*, ignore their application-wide policies) and return its own singleton instance.

VI. MAINTAINING A SINGLE RUNTIME

This section describes several technical details related to transformations intended to maintain the original application behavior.

A. Maintaining Pointers

Generally, since BREAKAPP starts its transformation from the object returned from a module (*e.g.*, `module.exports`), values *are* associated with a name: the name of the attribute associated with that value. However, not all values in messages include a meaningful name. For instance, a function can be anonymous and an object can just be a bytebuffer. To facilitate cross-compartment addressing, the child compartment maintains a hash table mapping object and function IDs (*e.g.*, SHA256 checksums) to their in-compartment pointers. These IDs can be thought as distributed, shared-memory pointers which RPCs include in their messages. Whenever it receives an RPC message, BREAKAPP on the child compartment looks into the table and routes freshly-deserialized arguments to the right function or object method.

B. DAG Structure and Reference Equality

The creation of object copies during transformation and serialization breaks reference equality. BREAKAPP takes care to preserve it. When an RPC leads to a new memory alias in the remote compartment, the return message from the remote compartment will include an `alias` entry containing the remote object ID. BREAKAPP on the child compartment then creates and returns a reference to an existing object. The same consideration must be extended to preserving reference equality for the root of the DAG between RPCs. A common pattern in many languages is to have methods that return `self`; such code would break if the return value of the RPC was a fresh copy of the method receiver.

C. Ordering

Messages get assigned a sequence number. Although communication primitives are reliable, messages should be received at the correct call order. For example, an asynchronous call to the printing function will be shown before the next call to the same function.

D. Calls to Constructors

Constructors, usually prefixed by the `new` keyword, slightly change the semantics of a function call: at the very least, new memory may need to be allocated. The RPC stub uses additional logic to detect this case.[1] If the function is indeed called with as a constructor, the RPC message has a special type signifying that the target function should also be called with `new`. The return value from a constructor is itself an object whose methods are RPC stubs as described earlier: the true object lies within its compartment.

E. Move vs. Copy Semantics

It is worth clarifying the distinction between values that are remotely referenced and ones that are copied to the parent compartment. When all nodes in the returned DAG are methods, they are transformed to RPC stubs referencing values that live within the remote compartment. State updates targeting such well-encapsulated modules or objects call directly into the remote object. When some nodes in the DAG are primitive values however, they result in deep copies of values. Writes to such values or the RPC stubs themselves[2] need to be detected and propagated to the original object.

[1] There are many possible ways of doing this; in JavaScript, the simplest one is to check the value of `new.target` within the wrapped function's scope.

[2] Whether this is allowed or not is a policy-specific question, discussed in Section V; here, we merely show how our mechanism *has* the ability to detect it.

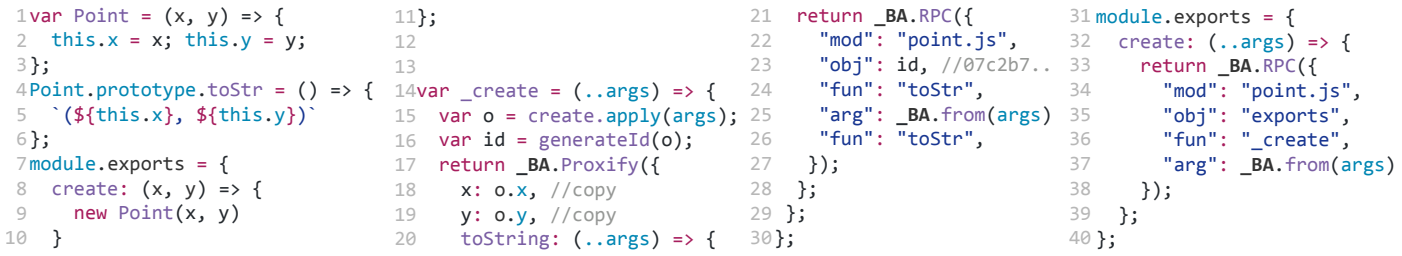

Fig. 3. A simplified example of BREAKAPP-related transformations. `_BA` corresponds to the BREAKAPP library.

To achieve this, we wrap the transformed output DAG with an interposition mechanism that provides reflection capabilities and gets invoked upon attribute accesses. A special BREAKAPP `Proxy` wrapper[3] detects and records changes to any of the object's properties. Property values that are themselves objects require nested proxies (Fig. 2). These state updates are compressed into changesets, and propagated lazily by piggybacking on future RPC calls.

F. The Class Hierarchy

In object-oriented programming languages, an object might invoke methods inherited from an object higher in the class hierarchy. These superclasses (or prototypes, for prototype-based languages such as Lua and JavaScript) might have been imported from a different module. A naive implementation of transformations to RPC stubs can then lead to a series of nested boundary-crossings until the outer RPC reaches its final destination. BREAKAPP detects class (prototype) hierarchy levels while traversing the DAG and crafts RPC stubs so that they immediately redirect to their final destination.

G. Native Functionality

Objects high in the prototype chain are supported natively. Functionality is either implemented internally in the runtime (*e.g.*, serialization and cryptography modules) or wraps OS-level subsystems (*e.g.*, networking and filesystem modules). In most cases, a copy of these objects can be found in the trusted copy of the runtime (see Section III-B) which BREAKAPP includes in the new compartment. Examples include modules such as `crypto`, `http-parse`, and `fs`, and globals such as timer functions and top-level objects.

There are cases when this is not possible, however. Specific global or pseudo-global[4] objects in the child compartment require redirection to the top-level compartment. Examples of such objects include `console` and `process` to refer to terminal output and process-level data, respectively.

If compartments live in different address spaces, writes to the child compartment's out and error streams must be transmitted to the top-level process. Upon first import, the system shadows `log`, `warn`, and `error` with such redirecting proxies. Similarly, it shadows stream input functions with functions that request this functionality from the top-level compartment, which sends the results back to the child.[5]

[3] Metatables in Lua or `reflect` in Java provide similar capabilities.

[4] Server-side JavaScript implementations make several objects that are not part of the EcmaScript specification available in the global scope, such as `process` and `console`.

[5] In practice, modules asking for top-level user input are extremely rare.

H. Garbage Collection

The standard runtime garbage collector (GC) cannot "see through" compartment boundaries to collect objects within translation tables. So, in addition to reflecting method calls between compartments using RPCs, BREAKAPP also propagates garbage collection events by adding a GC hook to every object that is the result of a transformation. When such an object is about to be collected, BREAKAPP sends a message to the child compartment to remove any references to this object.

Whole modules are more difficult to go out of scope for the GC to kick in and reclaim their memory. This is because there are multiple references to a module in the cache of the loaded modules. However, modules are often unloaded or reloaded manually, which should destroy or restart the child compartment. To maintain this behavior, BREAKAPP wraps the module cache structure, detects invalidations, and forces the child compartment to exit. Malicious modules cannot cause other modules to exit, because child compartments do not have access to other cache entries.

I. Monitoring

BREAKAPP interposes on inter-compartment communication, tracking the load placements and frequency of calls on each channel. It monitors the health (*i.e.*, crashed, not responding) of child compartments periodically and upon remote invocations. It takes curative actions based on the compartment's status (*e.g.*, restart, kill, or spawn more compartments). This is helpful in cases where the module within the compartment is launching a DoS attach or where asynchronous execution has lead to exceptions. Child compartments use OS primitives (*e.g.*, `SIGHUP` on Linux) to be notified upon parent exit.

J. Wrapping Up

Fig. 3 shows the result of a simple module after two stages of transformations. The first transformed the return value (`create`) of the module, and the second transformed the return value of a call into the module (a `Point` object). These transformations are done during runtime and captured only for illustrative purposes.

The left-most column contains most of the original module and its export statement. The right-most column exports a wrapper for `create` that serializes arguments and calls back to the original function. The middle two columns show the result of transforming a newly `_created Point` instance: `generateId` will store the object to a translation table, and return a remote reference. The transformed `toStr` will always

call back into the original object, whereas access to its `x` and `y` fields is `Proxy`ied through the interposition object.

VII. Other Languages

The work described so far is not tied to a particular programming language; whenever we use JavaScript and its ecosystem, it is only for illustrative purposes.

Interpreted languages are a particularly good fit for runtime compartmentalization. They expose a single function or function-like operator that takes care of locating a module, interpreting it, and exposing its interface in the caller context. Because all of this happens during runtime, the boundary detection that occurs at the import statement is conveniently unified with runtime compartment construction and code transformations. Moreover, the ability to (re-)bind different functions to the same variable names and interpose on object accesses further simplify things. We could see a straightforward implementation of BreakApp in languages such as Lua, Python, and Ruby.

Compiled languages do not enjoy these conveniences. The work done at runtime for JavaScript would need to be split across three phases: compile, link, and runtime. An implementation of BreakApp for compiled code would also face further challenges. First, modules may be linked and loaded statically or dynamically, which complicates the choice of how to divide work between the three phases. Second, type information may not be present at either compile time or run time in languages like C, thus complicating object introspection and marshalling. Finally, source code may not be available for all untrusted modules. Without source code, compiler-driven transformations become infeasible.

Some of these individual challenges have been recently addressed in the literature. For example, C-Strider [45] provides type-aware heap traversal for C programs. Concurrent with our work, researchers are just starting to tackle automated module isolation in compiled languages such as C [24] and Rust [54]. We believe this provides evidence that adapting BreakApp for compiled languages would be feasible, albeit nontrivial.

VIII. Implementation

This section describes a concrete implementation of our system targeting the JavaScript ecosystem, `breakapp`, along with some of its technical challenges and how they were addressed.

We built our prototype on top of Andromeda [56], a system aimed at simplifying the development of large-scale, distributed, general-purpose applications. The hosted version of Andromeda runs each node as a userspace process. Node management, synchronization, and communication are handled by Andromeda's built-in services. Low-level internals are handled by Node.js [11], a runtime that bundles (i) Google's V8, a fast JIT compiler, (ii) libUV, cross-platform wrappers for file-system and network operations, and (iii) several standard libraries, including OpenSSL used for hashing (SHA256). The `breakapp` package is open source and available for download via `npm install -g breakapp`.

Excluding all Andromeda code, the current prototype is approximately 2K lines of JavaScript. One third of them is for handling policies and other configuration parameters, and the rest supports transformations, serialization and low-level handling of different isolation primitives. For encryption, we use Dan Bernstein's NaCL library [4], compiled to JavaScript using Emscripten (adding another 2K LoC). Our implementation does not make use of any non-JavaScript features beyond the GC hooks mentioned in Section VI-H. For these, we use V8's weak finalizers with callbacks that fire when an object is about to be garbage collected.

Static Analysis There are several challenges related to policies that depend on the structure of the dependency tree. First, there is a one-to-one correspondence between source files and modules. There is little to no information during runtime about which packages file-level modules belong to. Second, deduplication causes the Node Package Manager to not install dependencies in a deterministic way. As a result, high-level granularity policies (*e.g.*, `LEVEL`, `GROUP`) can lead to different compartmentalization results depending on the package installation order. To mitigate these problems, `breakapp` statically analyzes information upon startup in order to make policy expressions meaningful. Starting from the leaves of the dependency tree, it creates a map from files to their source package. It also uses information from the various `package.json` files and the directory structure to infer a canonical dependency structure. This functionality is accessible through the tool's command line interface via `breakapp --create-map` (or `-c`), which makes it easy to use as a post-install hook for any package manager (*e.g.*, Yarn).

Non-blocking I/O A key challenge in the implementation of process and container isolation had to do with a conflict between Node.js's (i) non-blocking I/O and (ii) blocking `require` statement. To ensure that the `require` call returns *only after* the compartment has been created, `breakapp` in the parent compartment polls the filesystem constantly for a file that confirms that the child compartment has created the channel. Polling allows the system to actively check a number of different sources during each iteration (and timeout after a while). Since the channel is created *before* launching the new compartment, an alternative solution was to block on the channel for an ACK message. However, compartment creation might fail; thus, there is no guarantee that the parent compartment will not block indefinitely. Environments that expose any kind of preemptive multi-threading should not face similar issues.

IX. Evaluation

We use Node.js 6.9.1 (bundled with V8 v.5.1.281.84, libUV v.1.9.1, OpenSSL v.1.0.2j). Experiments were run on a Linux server with 512GB of memory and 160 hyper-threaded cores on Intel Xeon E7-8860 processors running at 2.27 GHz. We use Docker 17.06.0-ce for our container infrastructure.

A. Applicability

The techniques described in this paper are predicated on the hypothesis that applications today make extensive use of third-party packages. What are the modularity characteristics of JavaScript applications out in the wild? Table IV outlines the dependency characteristics of popular JavaScript programs

TABLE IV. FIVE CLASSES OF JAVASCRIPT PROGRAMS ALONG WITH THREE WIDELY USED INSTANCES AND THEIR DEPENDENCY CHARACTERISTICS.

	Application	Direct	Total	Files	Depth	ALoC	TLOC	TLoC/File
commands	cash	15	84	3554	5	1486	48540	13.84
	eslint	34	135	4689	6	187801	74893	39.97
	yo	30	301	5829	6	107713	106393	18.45
desktop	popcorn	46	765	34322	10	14304	411706	12.34
	twitter	10	120	4051	8	2514	165066	41.29
	atom	57	358	5252	9	15939	548642	107.1
mobile	hackernews	5	871	49406	10	309	317144	6.42
	mattermost	17	521	13672	8	6296	286388	21.37
	stockmarket	14	44	1985	5	2440	199119	101.48
server	express	26	42	217	3	10159	2261	54.93
	ghost	62	981	22029	9	42467	386676	19.35
	strider	64	659	10357	8	21090	303527	30.41
utils	chalk	3	4	9	2	217	10	18.44
	natural	3	3	193	1	12483	4116	81.51
	winston	6	6	83	1	4274	2326	79.52
average		26.13	326.27	10376.53	6.07	28632	190453.8	43.09

drawn from five different classes.[6] The table shows: (i) direct dependencies, referring top-level packages the application imports, (ii) total dependencies, including all packages in the dependency graph, (iii) total of file-level modules, (iv) the depth of the dependency tree, (v) non-third-party lines of code, *i.e.*, lines the author wrote, (vi) total lines in third-party, imported code, and (vii) the average lines of code per third-party file-level module.

Third party code is a non-trivial portion of today's applications. In our sample set, imported code is on average 4 times larger than homegrown; but the ratio is much worse for large applications (1:120 for hackernews vs. 2:1 chalk). Different applications spread third-party code differently. For example, in mobile applications, more than 99% of their third-party code comes from a single package—the mobile framework in use (*e.g.*, Ionic, ReactNative). Server-side applications feature the largest amounts of total third-party counts, followed immediately by desktop applications.

Direct module counts, the boundaries of trust between the code that a developer writes and its third-party dependencies, are somewhere between 2 and 65. These numbers highlight the minimum number of compartments (average: 26.1). More fine-grained compartmentalization at the level of individual packages requires an order of magnitude more compartments (average: 326.2). Since there is a one-to-one correspondence between files and modules, file-level compartmentalization *is* possible but would require 1-2 orders of magnitude more compartments (*e.g.*, popcorn has more than 10K JavaScript files). Interestingly, analyzing more than 1K imports (translating to more than 100K file-level modules) reveals an average ratio of 43 lines of code per file, exceeding our expectations for least-privilege decomposition.[7]

B. Security

Does the system mitigate vulnerabilities (both discovered and hypothetical) similar to the ones outlined in Section III?

[6] We do not discuss client-side web apps, since the emphasis of our work is language-agnostic, system-level decomposition (it just happens to use JavaScript, historically created for client-side web development). To address the reader's curiosity, however, here are some numbers: 1060 modules for apple.com, 1050 modules for the mobile version of reddit.com, and 365 modules for keybase.io.

[7] As a point of comparison, Minix 3 [21], a modern microkernel that championed least-privilege separation, comes with userspace servers on the order of thousands of lines of code.

TABLE V. VARIOUS REAL (TOP) AND HYPOTHETICAL (BOTTOM) VULNERABILITIES, AND THE POLICIES USED TO MITIGATE THEM.

Package	Ver.	Type	R/H	Mitigation
qs	6.0.0	introspection, poisoning ⓒⓓⓖ	R	sandbox
serialize-to-js	0.4.8	eval ⓕ	R	sandbox
fernet	0.0.9	timing attack ⓟ	R	sandbox
uri-js	2.1.1	denial of service ⓚⓞ	R	process
libxml	0.16	unsafe extension ⓖ	R	process
hostr	2.3.2	read file-system ⓘ	R	container
glob.js	—	global variables ⓐⓑ	H	sandbox
this.js	—	context ⓒ	H	sandbox
mod.js	—	module cache ⓜ	H	sandbox
argv.js	—	process args ⓝ	H	process
env.js	—	user environment variables ⓝ	H	process
stack.js	—	inspect call stack ⓔ	H	process

Table V presents twelve vulnerable modules, along with the compartment types used to mitigate their effects. The first six are public packages, and their vulnerable version is shown in the second column. For these packages, we use the exploit attached to the original vulnerability report. The last six are hypothetical vulnerabilities; although we were not able to find any packages with these specific vulnerability types in any vulnerability databases, we know they are possible to construct. All of them can be found in the online appendix.

Most of the first six packages can be used for a number of attacks. For example, `serialize` makes use of the Turing-complete `eval` function; user code can access global variables, patch system APIs, and inspect loaded modules. Launching it in a fresh V8 sandbox (`SBX`) defends against inspection and patching of the main program's data and structures. Since it is not written in C, it cannot forge pointers to bypass the language's safety features; therefore, launching a process would only add protection against denial-of-service attacks (*e.g.*, using `eval` to start an infinite loop), process arguments, and the shared environment. This highlights the core benefit of policies: they let developers specify what they care about based on *their* application structure and needs.

Over half of the problems can be mitigating simply by using sandbox (`SBX`). Defending against `glob.js` and `this.js` required explicit whitelisting of references (`CONTEXT`). Shielding against snooping the environment was possible via process-level isolation (`PROC`) and selective shadowing of environment variables and process arguments. Such shadowing (*e.g.*, `ENV`) creates an artificial copy of selected variables right after launching the compartment but before loading the module. Since the module system is grafted atop

TABLE VI. TOP-LEVEL CHARACTERISTICS OF PACKAGES CHOSEN FOR PERFORMANCE EVALUATION; CALLING FUNCTIONS OR CONSTRUCTORS GENERATES FURTHER DAG NODES.

Package	PLoC	Files	Nodes	Depth	Fan-out	Functions	Highlights
http-verbs	29	1	28	2	27.0	0	small, flat, string-to-string mapping; stress interposition proxies
left-pad	52	1	1	1	1.0	1	small function; stress RPCs
cash	451725	10839	75	7	314.0	49	large libraries; system calls; stress loading time
chalk	145706	9630	5	3	5.3	2	stress builder objects/cascading calls; encoding
debug	554746	8657	34	4	51.3	14	stress non-functional updates; varargs; console output back to parent
ejs	59396	4950	25	4	12.0	11	extensive, pure, testing fixtures
tweet-nacl	94686	5387	54	5	40.8	42	crypto; stress processing; (incl. encoding/decoding)
dns	4826	1	60	3	34.0	16	built-in module; async calls

TABLE VII. COMPARTMENTALIZATION COSTS: COMPARTMENT STARTUP TIMES.

Compartments	Standard	Sandbox	Process	Container
5	4.3ms	12.9ms	342.5ms	5.9s
50	30.2ms	76.6ms	3.2s	32.8s
500	136.4ms	524.7ms	35.2s	332.2s
5K	1.7s	7.8s	362.4s	3330.5s
abs. / +1 cmpt	0.3ms	1.5ms	72ms	666.1ms
rel. / +1 cmpt	(baseline)	5×	240×	2220×

TABLE VIII. COMPARTMENTALIZATION COSTS: THROUGHPUT AND LATENCY.

Comp/nts	Function	Pipe	UDS	TCP
5	192.3GB/s	18.3GB/s	149.5MB/s	158.1MB/s
	6.5ns	1.3–1.4ms	17.8–73.8ms	17.7–36.6ms
50	157.1GB/s	17.5GB/s	127.0MB/s	134MB/s
	90.18ns	11.6–13.2ms	244.5–536.6ms	210.3–566.8ms
500	46.5GB/s	3.6GB/s	16.4MB/s	20.9MB/s
	294.3ns	154.3–160.3ms	3.71–11.95s	6.5–15.6s

V8, separate V8 sandboxes do not have access to already loaded modules if not explicitly shared during sandbox construction. `mod.js` required whiltelisting the `module` module with a *fresh* module cache. The call stack and event queue are shared between different V8 sandboxes; disabling access to the call stack for `stack.js` requires a separate process. Mitigating timing channels for `fernet` required only a sandbox (`SBX`) and a constant minimum response time (`TIMER`). Mitigating `url-js`'s DoS problems required processes (`PROC`), replication (`REPLICAS`), and a special scheduling policy (`SCHED`). We will see the details in Section IX-E.

C. Micro-benchmarks

What are the overheads of different isolation mechanisms related to policies? Tables VII and VIII highlight the costs of starting up new compartments and crossing compartment boundaries under various configurations.

For the first experiment (Table VII), we minimize the effects of module sizes by making modules return a single integer and launch compartments sequentially. *Standard* is how the vanilla module system works: it looks up a module on the filesystem using a resolution algorithm, wraps it so that its global variables do not leak to the outer context (and to provide some global-looking variables, *e.g.*, filename), and evaluates the code in the current context. *Sandbox* creates a new V8 context for each module and selectively whitelists shared variables from the parent context. *Process* and *Container* use OS processes and Docker containers to isolate compartments between each other, resulting in higher startup costs.

For the second experiment (Table VIII), we process an in-memory (`/dev/shm/`) stream of 1GB using a linear pipeline of mostly-empty stages. Pipeline stages only flush some timing metadata when they detect the end of a stream. Streaming starts only after all connections have been established (*i.e.*, no TCP handshake costs included).

D. Performance

What is the performance overhead of spawning modules in their dedicated compartments using BREAKAPP? We opt for single-module, single-compartment setups over multi-level compartmentalization to zoom into the exact sources of overhead under various configurations. We use a diverse set of eight modules to isolate and account for various different behaviors (*e.g.*, interposition, RPCs, *etc.*). These modules are running under the Node.js framework, serving HTTP requests. Table VI summarizes the source-level aspects of the modules used. Generally, lines of code and files correlate with import times; number of nodes, depth, average fanout correlate with interposition transformation costs; and the number of functions correlates with the function-to-RPC transformation costs — a much heavier transformation compared to interposition proxies.

Fig. 4 breaks down startup latencies into various sources (*e.g.*, transformations, interposition *etc.*) between the main four different compartment types. `IPC` was set to `TCP` to account for its heavy setup period (yellow segment; 17.6–35ms); this choice affected communication latencies too (purple segment; 11.6–24.8ms). To ease comparisons, we did not include system-level costs of spawning each compartment; these are presented in Table VII. **Startup costs are dominated by the number of modules (*i.e.*, files) read from the file system.** A good example is `cash` where importing all the sources takes 798.2–1049.1ms compared to 138.5ms of launching a new process and 847.9ms of launching a new container. For smaller modules such as `http-verbs` and `left-pad` the overhead of launching the compartment is more pronounced, but these modules tend to be used for long-running processes (*e.g.*, web servers); in these cases, a startup time of few hundred milliseconds gets amortized over a period of days. Transformation overheads were generally on the order of 0.2–1.3ms for the addition of interposition proxies and 0.5–6.1ms for all the rest.

Fig. 5 shows their execution latencies. `IPC` is set to `PIPE`; each IPC segment on the figure includes the overhead of a serialization and deserialization pair. To account for a more realistic setup, modules are loaded as part of a larger "no-op" HTTP application which does not do any other processing beyond calling the dedicated module. Latencies are averages over 1K requests following 100 warmup requests. Some of the more processing-heavy modules (*e.g.*, `left-pad`, `tweet-nacl`) were tested under different types of workloads: a small 5B workload and a larger 5MB one.

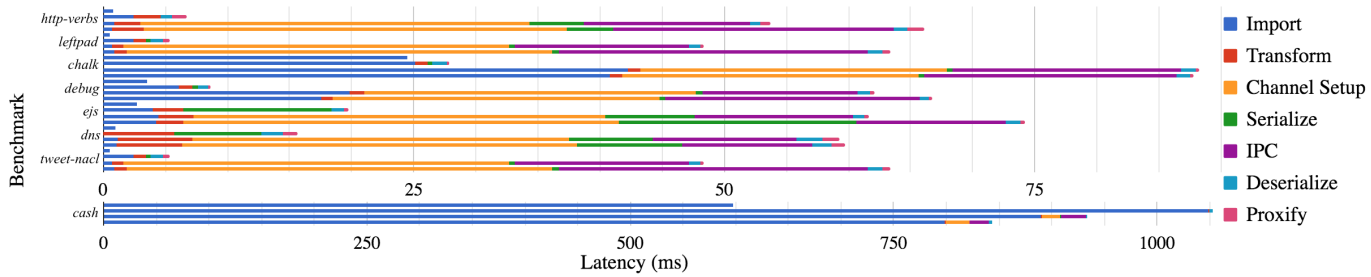

Fig. 4. Startup breakdown of eight packages under various configurations: vanilla, sandbox, process, and container.

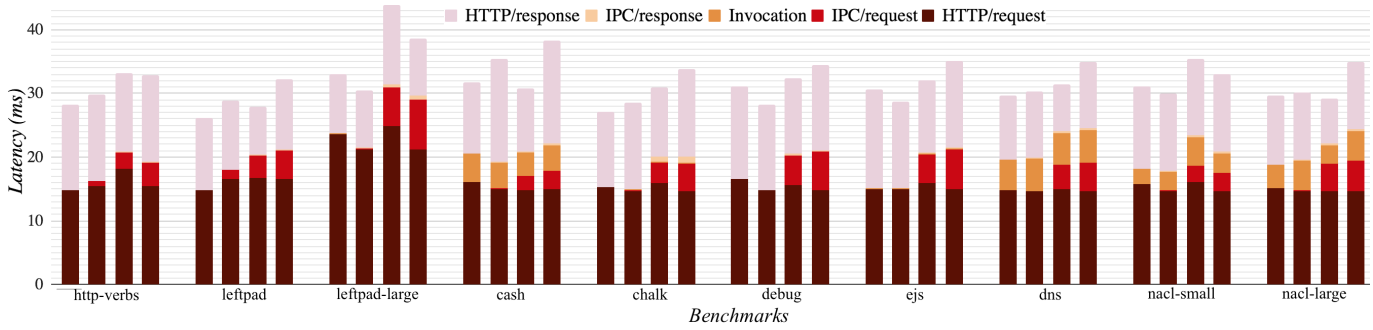

Fig. 5. Latency breakdown of 10 different workloads with four compartment types for each: vanilla, sandbox, process, container.

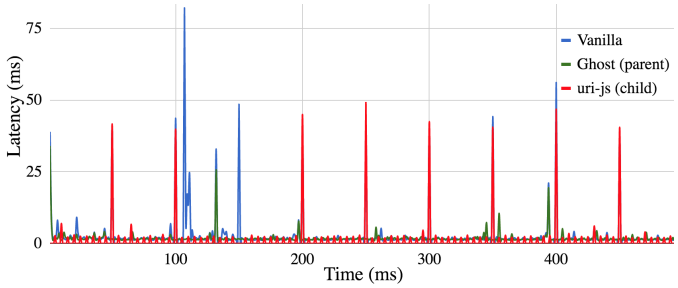

Fig. 6. Denial-of-service attack against a blogging platorm: (i) vanilla setup (blue); (ii) two process-level compartments (green + red).

Since HTTP request and response handling in Node.js dominate latency, compartmentalization overheads (IPC request and response) account for a small part of the overall latency. **Even in the case of the heavier compartment types, they are responsible for 2 to 15% of the overall latency.** The vast majority of this overhead is concentrated on the calling side. Returning results is much cheaper because the return values in our experiments were typically smaller.

The overhead of proxy (interposition) objects is barely visible in these plots, and generally much smaller than initially expected. To understand the costs of object proxying better, we created objects with a fanout of 12 for 8 levels (*i.e.*, with 12^8 internal nodes — roughly .5GB memory footprint). Traversing one million 12-hop random paths to access properties of the object took 167.2ms on the original object and 595.7ms on the proxy-augmented object (*i.e.*, all calls went through the proxy object). To put these numbers into perspective, the object allocation took nearly 16 seconds, meaning that **applications will likely hit other bottlenecks before the overhead of interposition becomes noticeable.**

E. Denial of Service Mitigation

Can BREAKAPP be used to mitigate DoS attacks? Fig. 6 shows latencies of issuing 500 HTTP requests to a slightly modified Ghost server where `url-js` is susceptible to DoS attack. The workload consists of 90% benign read requests of various posts; 8% benign search requests; and 2% malicious search requests. Malicious search requests block the event loop for 40–60ms.

In the first configuration (blue line), all modules run in a single process. Read requests reaching the server right after a malicious search query get delayed significantly or, at higher rates (not shown here), timeout. In the second configuration (green line: 90% reads; red line 10% queries), the `url-js` module runs in a separate compartment with a policy of `PROC` and `PIPE`. Although it takes slightly longer to process search requests (+1ms on average), malicious search requests do not block benign read requests: it is only the subset that goes through the search functionality that remains paralyzed by the DoS attack.

In a different experiment, we increased both the number of malicious requests as well as their latency. Due to monitoring, `breakapp` is aware that some requests are taking significantly more time than expected. By examining the input to the most recent RPC, it can distinguish between problematic inputs and non-problematic ones. We experimented with four `ON_FAIL` policies: (i) shut the child compartment down and report; (ii) restart the compartment; (iii) spawn a new replica and use a scheduling policy (*e.g.*, round robin) to schedule RPC calls to these replicas; or (iv) pushback based on recently-seen inputs. We crafted careful "asymmetric" attacks where a small number of malicious requests blocks the event loop for extended periods of time.

The combination of multiple `utl-js` replicas and caching

of results from requests that take longer than 0.5s with a 30s age timeout was successful at mitigating them. This was a serious improvement over the previous setup: we were not able to saturate the system without generating *additional* malicious strings.

In our final experiment, we overrode the round-robin scheduling policy during runtime by passing a function that implements priorities. BREAKAPP split requests into 100 different queues based on the length of the input string. Benign, small input strings (*i.e.*, nine out of ten requests) always had available resources.

Further mitigation is possible. Parallel spawn of (i) 500 compartments took 2.82 seconds, and of (ii) 5K compartments took 24.3 seconds, indicating good elasticity characteristics until system administrators act upon notifications.

X. RELATED WORK

Supply Chain Attacks Our concerns about large-scale reliance on loose supply chains are echoed by both academia [30], [25] and industry [28], [8], [51]. Academic studies have shown the increasing risks of the reliance on third-party code (although they generally do not consider the scope of problems we do in Section III). Several recent companies [36], [38], [51] provide third-party module assurances by having more people audit and recommend packages in the wild or crawl public repositories for open vulnerabilities. In practice, they do not offer any guarantees similar to compartmentalization, but can be used complementary to our work: users (or libraries that are built on top of BREAKAPP) can use these recommendations to choose which modules to quarantine. Package managers have added support for locking dependencies between deployments [37]. However, this does not necessarily rule out extant problems; on the contrary, users forego valuable bug and vulnerability fixes, while experiencing a more convoluted dependency management.

JavaScript Isolation In the case of JavaScript specifically, much effort has gone into client-side compartmentalization (*e.g.*, execution isolation [33], object capabilities [34], sandboxing [53], [2] information flow control [52]). These works focused primarily on client-side, web-based setups which are different from our focus in many ways: isolation primitives (*i.e.*, iframes), origin (*i.e.*, explicit sources), threat model (*i.e.*, no C/C++ modules; no valid access to "/etc/passwd"), compartment numbers (*i.e.*, few), and developer effort (*i.e.*, manual annotations or rewrite).

Microservices Microservice architectures, a style for building server applications as sets of loosely-coupled components [17], [35], are often touted as enabling fine-grained, Least-Privilege decomposition inspired by the Unix philosophy. Even more so, lambda architectures [14], [20] are emerging as a lighter-weight, evolutionary step beyond microservices that use runtime contexts to offer improved elasticity. In practice, however, both are vastly more coarse-grained than the applications shown here, with each microservice usually built on top of hundreds of packages similar to the server-side applications outlined in Table IV. Moreover, (i) communication between services is request-response style and usually explicitly exposed to the application,[8] (ii) decomposition is a manual process that requires a careful design process (including agreeing on the interfaces) prior to development. These are antithetical to our technique that hides the underlying compartment boundaries.

Containerization Many other system-level sandboxing primitives can be used (*e.g.*, SELinux [29], AppArmor [3]). We experimented with Docker [32] primarily because it is becoming an industry standard: most users are expected to use (and seek numbers about) Docker more than any other container infrastructure. Coarse-grained compartmentalization, however, such as by wrapping a language runtime with a layer of virtualization, is ill-equipped to address risks described in Sections II-A and III: a malicious module is still able to access trusted state, exfiltrate data, or launch a DoS attack.

System Decomposition There is a long history of alternative system structures with a focus on least privilege decomposition [44] and, more generally, separation of concerns [13] (*e.g.*, microkernels [22], [27], [21], capability systems [26], [48], and separation kernels [42]). Increasing security requirements brought decomposition to the foreground [40], culminating with systems such as Crowbar [5] and SOAAP [19] that assist programmers into decomposing applications into multiple compartments with reduced privileges. Other systems have focused on abstractions and mechanisms that allow efficient separation [12], [58], [57]. Despite their advances, systematic adoption has been impeded by the lack of automation [30], the primary focus of BREAKAPP.

Unsound Program Transformations There is a recent emergence of unsound program transformations [41], [39], [49], [50] to mitigate failures and attacks via self-healing. Such transformations change the semantics of the original program in principled ways. Our work can be seen as a set of potentially unsound transformations at the module boundaries: users can decide how to break the semantics of the program by choosing which behaviors to disable. However, our transformations are proactive rather than reactive. They are also based on the assumption that there are cases in which a legacy program runs the risk of breaking *either way*: developers can choose whether it will be due to third-party modules (unprincipled way) or due to altered semantics (principled way).

XI. FUTURE

BREAKAPP is the first step towards the goal of having applications with many, potentially risky, third-party modules be more secure than their monolithic counter-parts specifically built for security. There is ample potential for improvement, that can build upon the core model and mechanism of module-driven compartmentalization.

Automating Security Policies One direction is automatically inferring policies that improve an application's security without breaking its functionality. This clearly depends on a threat model, but our experience with BREAKAPP shows that there is plenty of room for hardening application security before starting to break compatibility. The challenge here is to detect "good enough" policies: (i) static analysis is not trivial in

[8] It usually relies on HTTP — much more heavyweight than the channels described in this work.

an environment with zero type annotations and compiled, unsafe modules from multiple languages; (ii) dynamic analysis requires tracing pre-runs and does not have clear cut-offs (*i.e.*, how do we know we are not tracing adversarial module behavior?).

Optimizing Performance BREAKAPP's mechanisms execute at program runtime, raising the tantalizing possibility of *dynamically* separating and coalescing modules, driven by runtime profiling feedback. The challenge here is to formalize the requirements as an online optimization problem: security benefits are difficult to quantify and the overhead budget has multiple dimensions (*e.g.*, latency, throughput *etc.*)

Incremental Hot-Patching A further direction is patching applications by updating individual modules during runtime. Given isolated modules from BREAKAPP, compartments can be used to enable incremental, restart-free updating of applications, which can benefit security and performance. Significant challenges include migrating state of individual modules from the old to the new version and the frequency of updates in applications comprising of hundreds of modules.

XII. CONCLUSIONS

Third-party, untrusted modules have simplified application development at the cost of security and reliability. This work demonstrates, with a working prototype, that it is possible to take advantage of third-party modules to offer significant automation improvements to compartmentalization. The core technique can spawn a module in its own compartment while maintaining its interface intact. Significant automation is focused at three levels: (i) transformations for spawning modules into their own compartments, (ii) declarative policy expressions, where each policy can have multiple effects at various different levels, (iii) transformations and interposition to maintain the illusion of a single runtime. A concrete implementation for JavaScript shows that BREAKAPP simplifies security hardening of existing systems while maintaining acceptable performance levels.

ACKNOWLEDGMENTS

We would like to thank Athur Azevedo de Amorim, Andreas Haeberlen, Cătălin Hriţcu, Björn Knutsson, Benjamin C. Pierce, John Sonchack, Nik Sultana, and the anonymous reviewers for helpful feedback. This research was funded in part by National Science Foundation grant CNS-1513687. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] "CVE-2016-2537." Available from MITRE, CVE-ID CVE-2016-2537., 2016. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2537

[2] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "Jsand: Complete client-side sandboxing of third-party javascript without browser modifications," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 1–10.

[3] M. Bauer, "Paranoid penguin: Apparmor in ubuntu 9," *Linux Journal*, vol. 2009, no. 185, p. 9, 2009, accessed: 2016-09-30.

[4] D. J. Bernstein, B. Van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "Tweetnacl: A crypto library in 100 tweets," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2014, pp. 64–83.

[5] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 309–322.

[6] F. Brown, A. Mirian, A. Jaiswal, A. Notzli, and D. Stefan, "SPAM: a Secure Package Manager," 2017. https://cseweb.ucsd.edu/~dstefan/pubs/brown:2017:spam.pdf

[7] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[8] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 516–519.

[9] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:5.

[10] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 3–3.

[11] R. Dahl and the Node.js Foundation. (2009) Node.js. Accessed: 2017-06-11. https://nodejs.org

[12] U. Dhawan, A. Kwon, E. Kadric, C. Hritcu, B. C. Pierce, J. M. Smith, A. DeHon, G. Malecha, G. Morrisett, T. F. Knight *et al.*, "Hardware support for safety interlocks and introspection," in *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 1–8.

[13] E. W. Dijkstra, "On the role of scientific thought," in *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 60–66.

[14] M. Eriksen, "Your server as a function," in *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, ser. PLOS '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:7.

[15] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to javascript," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13. New York, NY, USA: ACM, 2013, pp. 371–384.

[16] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Professional, 1999.

[17] M. Fowler and J. Lewis. (2014) Microservices. Accessed: 2015-02-17. http://martinfowler.com/articles/microservices.html

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European Conference on Object-Oriented Programming*. Springer, 1993, pp. 406–431.

[19] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with soaap," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1016–1031.

[20] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016.

[21] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 80–89, Jul. 2006.

[22] M. J. Accetta, R. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Wayne Young, "Mach: A New Kernel Foundation for UNIX Development," in *USENIX Summer Technical Conference*. Usenix, 06 1986, pp. 93–113.

[23] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 567–581.

[24] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, "Sandcrust: Automatic sandboxing of unsafe components in rust," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, ser. PLOS'17. New York, NY, USA: ACM, 2017, pp. 51–57.

[25] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2 2017.

[26] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[27] J. Liedtke, K. Elphinstone, S. Schonberg, H. Hartig, G. Heiser, N. Islam, and T. Jaeger, "Achieved ipc performance (still the foundation for extensibility)," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE, 1997, pp. 28–31.

[28] J. Long. (2015) Owasp dependency check. Accessed: 2017-02-17. https://www.owasp.org/index.php/OWASP_Dependency_Check

[29] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 29–42.

[30] M. Maass, "A theory and tools for applying sandboxes effectively," Ph.D. dissertation, CMU, 2016.

[31] S. McConnell, *Code complete*. Pearson Education, 2004.

[32] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[33] J. Mickens, "Pivot: Fast, synchronous mashup isolation using generator chains," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 261–275.

[34] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized javascript," *Google, Inc., Tech. Rep*, 2008.

[35] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2015.

[36] Node Security. (2016) Continuous security monitoring for your node apps. https://nodesecurity.io/. Accessed: 2017-01-01.

[37] npm, Inc. (2012) npm-shrinkwrap: Lock down dependency versions. Accessed: 2017-02-03. https://docs.npmjs.com/cli/shrinkwrap

[38] E. Oftedal *et al.* (2016) Retirejs. Accessed: 2017-05-18. http://retirejs.github.io/retire.js/

[39] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 87–102.

[40] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 16–16.

[41] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr., "Enhancing server availability and security through failure-oblivious computing," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 21–21.

[42] J. M. Rushby, "Design and verification of secure systems," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 12–21.

[43] S. Saccone. (2016) npm fails to restrict the actions of malicious npm packages. https://www.kb.cert.org/vuls/id/319816. Accessed: 2017-06-05.

[44] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[45] K. Saur, M. Hicks, and J. S. Foster, "C-strider: Type-aware heap traversal for c," *Softw. Pract. Exper.*, vol. 46, no. 6, pp. 767–788, Jun. 2016.

[46] I. Z. Schlueter *et al.* (2010) Node package manager. Accessed: 2017-02-17. https://npmjs.com

[47] N. Seriot, "Parsing JSON is a minefield," 2016. http://seriot.ch/parsing_json.php

[48] J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: A fast capability system," in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, ser. SOSP '99. New York, NY, USA: ACM, 1999, pp. 170–185.

[49] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: Automatic software self-healing using rescue points," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 37–48.

[50] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a reactive immune system for software services," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 11–11.

[51] Snyk. (2016) Find, fix and monitor for known vulnerabilities in node.js and ruby packages. https://snyk.io/. Accessed: 2017-05-18.

[52] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining javascript with cowl," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 131–146.

[53] J. Terrace, S. R. Beard, and N. P. K. 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)*. Boston, MA: USENIX, 2012, pp. 95–100.

[54] S. Tsampas, A. El-Korashy, M. Patrignani, D. Devriese, D. Garg, and F. Piessens, "Towards automatic compartmentalization of c programs on capability machines," in *Workshop on Foundations of Computer Security 2017*, ser. FCS'17, 2017, pp. 1–14.

[55] N. P. Tschacher, "Typosquatting in programming language package managers," Bachelor Thesis, University of Hamburg, March 2016.

[56] N. Vasilakis, B. Karel, and J. M. Smith, "From lone dwarfs to giant superclusters: Rethinking operating system abstractions for the cloud," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015.

[57] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume ii: Privileged architecture version 1.7," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49*, 2015.

[58] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[59] A. G. Williams. (2016) Changes to npm's unpublish policy. http://blog.npmjs.org/post/141905368000/changes-to-npms-unpublish-policy.

[60] S. Yegulalp. (2016) How one yanked javascript package wreaked havoc. http://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html.