Towards Fine-grained, Automated Application Compartmentalization

Nikos Vasilakis
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
nvas@seas.upenn.edu

Ben Karel
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
karel@seas.upenn.edu

Nick Roessler
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
nroess@seas.upenn.edu

Nathan Dautenhahn
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
ndd@cis.upenn.edu

André DeHon
University of Pennsylvania
200 S. 33rd St.
Philadelphia, PA 19104
andre@acm.org

Jonathan M. Smith
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
jms@cis.upenn.edu

Abstract

The rise of language-specific, third-party packages simplifies application development. However, relying on untrusted code poses a threat to security and reliability.

In this work, we propose exploiting module boundaries – and the general trend towards more and smaller modules – to achieve fine-grained compartmentalization. Automated transformations can hide compartment boundaries and minimize developer effort. Optional policy expressions can decouple security assumptions at development time from requirements during composition and runtime. Using JavaScript's flourishing ecosystem, we discuss a wide range of risks and sketch how the use of language-level solutions coupled systemic mechanisms can protect against them.

CCS Concepts • **Security and privacy** → *Software and application security*; *Denial-of-service attacks*; • **Software and its engineering** → *Modules / packages*;

Keywords Least-Privilege Separation, Compartmentalization, Packages, Modules, Security

ACM Reference format:
Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2017. Towards Fine-grained, Automated Application Compartmentalization . In *Proceedings of PLOS'17, Shanghai, China, October 28, 2017,* 8 pages.
https://doi.org/10.1145/3144555.3144563

PLOS'17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5153-9/17/10.
https://doi.org/10.1145/3144555.3144563

	GH Repos	Package Repo	Size	Growth
JavaScript	931,493	npm, ++	530,050	507/day
Java	778,001	Maven Central	194,954	140/day
Ruby	569,180	RubyGems, ++	134,764	33/day
Python	454,042	PyPi, +	113,602	72/day
PHP	408,210	Packagist, ++	149,699	137/day

Table 1. Five popular programming languages and information about the size and growth of their package repositories (Snapshot: Aug. 1^{st} 2017). "+" symbols indicate more package repositories (not counted here).

1 Introduction

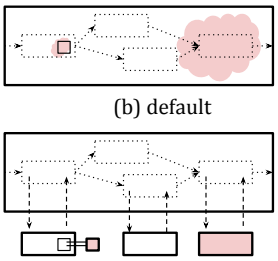
Composing software has changed significantly in scale, process, and basis for trust. Software such as the Linux kernel had many people focused on the quality and security of a single, large codebase [36]; even such a cohesive effort failed to prevent a slate of vulnerabilities [6, 8].

Today, however, programmers make increasing use of third-party modules from language-specific repositories. These repositories contain tens of thousands of packages (Table 1) from thousands of different authors. As a result, applications can have hundreds of third-party dependencies (Table 4) that execute without meaningful privilege separation or isolation beyond type safety guarantees. Although using code from modules minimizes developer effort (*e.g.*, reduced development and maintenance costs), it exposes the system to security risks (Table 2).

Further problems worsen these risks. Understanding the internals of a complex package and verifying that it will not behave in unintended ways [12, 29] are both difficult tasks. The popularity of certain packages allows vulnerabilities deep in the dependency graph to cause widespread difficulties [2, 5, 20, 27, 46, 50]. Vulnerabilities are becoming hard to eradicate, since (i) some updates are fetched automatically [39], and (ii) module unpublishing has become difficult in order to avoid breaking dependency chains [49]. In the

```
1 var dbc = require("./dbc.json");
2 var ejs = require("ejs");
3 function (req, res) {
4   var m = require("minimatch");
5   var s = m.test(/d/, req.body);
6   // do sth with result and db
7   res.end()
8 }
```

(a) simplified server-side code



(b) default

(c) w/ BreakApp

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

face of such challenges, the emerging practice is to avoid packages not endorsed by security advisories [7, 25, 43].

Instead of merely reacting to announced vulnerabilities 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 while offloading enforcement to the operating system (*e.g.*, address space isolation, LXC/namespaces, scheduling). We propose a drop-in replacement of a language runtime's module system, BreakApp, that uses module boundaries as guides to draw the lines between compartments.

BreakApp is centered around a *parametrizable transformation* technique that spawns modules in their own dedicated compartments during runtime. Automated transformations hide compartment boundaries by converting function calls to remote procedure invocations. Optional *runtime policy* expressions fix aforementioned parameters, effectively decoupling assumptions made during module development from requirements present during module composition. Since a single module can be used by several different types of applications, it is important to let the application developer choose which module behaviors to disallow. Policies also improve BreakApp's performance, since they allow programmers to customize the provided functionality on a per-import basis.

BreakApp does not require any annotations, does not require any trace (pre-)runs, and does not rewrite any source code. Moreover, policy expressions are backwards-compatible with existing codebases and forwards-compatible with vanilla module systems. As a result, the system lowers potential barriers to widespread adoption and makes incremental security retrofit in existing systems possible.

2 Problems

Consider a widely deployed, open source publishing platform written in JavaScript such as Ghost [14]. At its core, this application is an HTTP server that serves HTML generated from Markdown files, with the ability to edit and search documents, attach hypermedia, and include comments. Ghost has 62 top-level dependencies; counting recursive imports, the total jumps to 981 packages.

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
Env/Args Exfiltration	crossenv, babelcli, ffmepg, ++

Table 2. Eight major vulnerability classes and specific instances of packages available on npm [41], one of the main JavaScript package repositories; "++" indicates that many more packages with similar problems exist.

Fig. 1 – (a, b) presents a highly simplified version of Ghost highlighting typical module usage in modern applications. Different 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?

Example 1 As a simple example, suppose that Ghost uses `ejs` for template generation (Fig. 1 (a), line 2). A malicious version of this module could try to access the database credentials by any of the following ways: (i) attempt to read the global, singleton `dbc` object, (ii) import itself the `dbc.json` config file from the file system, or (iii) access the loaded config module directly through the module cache. In a conventional setup, all three are possible, mainly because it is difficult to distinguish these illegitimate behaviors from legitimate ones: any function can reach into global scope, any module can import built-in modules to read the file system, and any part of the program has direct access to cached modules for performance reasons.

Example 2 As a more interesting case, suppose Ghost provides search functionality (on line 5) using the `minimatch` module, which will necessarily be supplied user-generated strings. Even if `minimatch` itself is benign, a malicious user can launch a RegEx DoS attack by sending pathological regular expressions [9]. Given that many high-performance implementations follow an event-driven, cooperative concurrency model, a problematic search query can cause the application to stop accepting requests until the pathological request completes.

These two examples scratch only the surface of what is possible due to problematic modules. Table 2 summarizes a few vulnerabilities discovered in widely-used JavaScript

packages. Such flaws can be attributed to a number of factors: (i) common features of dynamic programming languages (*e.g.*, call stack inspection, reflection capabilities, monkey patching), (ii) language deficiencies (*e.g.*, in JavaScript: default-is-global, prototype poisoning, mutability attacks), (iii) implementation-specific choices (*e.g.*, event-driven implementations, cooperative multitasking, module system cache), (iv) authority considerations where any part of the application can read or write like any other (*e.g.*, read `process.env` or `process.args`, write to the filesystem or network), and finally (v) use of "native" modules[1] that nullify the safety guarantees provided by a high-level programming language.

3 System Overview

BreakApp is a *backwards-compatible, drop-in replacement* of the language's module system. It identifies when to spawn a new compartment (*e.g.*, for each module) and then lets the operating system handle issues such as isolation, communication, and scheduling. It can either be imported as an application-specific module (where it must be loaded before any untrusted code) or replace the system-wide built-in module system. It primarily consists of three interrelated tasks (Fig. 1 – (c)): (i) upon import, setup a new compartment along with a communication channel between the two; (ii) transform subsequent function calls to remote procedure calls (RPCs); (iii) periodically monitor compartments for health and status updates. For now, we can think of compartments as processes and communication channels as FIFO pipes or Unix domain sockets, but we will soon discuss several different types when talking about policies.

Compartment Setup BreakApp first dynamically replaces (*viz.*, "shadows") functions responsible for importing modules (*e.g.*, `require("ejs")`) with thin interposition wrappers: whenever the program executes a function that imports a new module, control jumps to BreakApp. If a module with this name is not already loaded in its own compartment, BreakApp (i) creates a new *child* compartment that imports only this module, and (ii) sets up a new communication channel between the two. The module system on the child's side inspects the direct acyclic graph (DAG) object returned by the import, and recursively replaces each node with a wrapper:

primitive values are copied unmodified and wrapped with an interposition mechanism that records changes.

function values become RPC stubs, which serialize arguments, send them through the channel, and wait for the results.

mutable values have their getter and setter functions replaced with RPC stubs.

[1] Other examples of unsafe modules in safe languages include `unsafePerformIO` in Haskell, `unsafe` blocks in Rust, C/C++ modules in Ruby and Python, and foreign function interfaces (FFI) in Lua and Racket.

Policy	Explanation
Type	Compartment type (*e.g.*, context, process)
IPC	Communication type (*e.g.*, FIFO, UDS, TCP)
Context	Whitelist pointers to parent context
Instantiate	Fresh compartment for each import
Replicate	Multiple replicas; schedule round-robin
OnFail	Action upon failure (function)
MinTime	RPC results available only after *min* time
Group	Group modules in a single compartment
Preload	Create proactively instead of lazily
Trust	Whitelist allowed modules
Doubt	Blacklist disallowed modules
Composition	How to combine policies in conflict

Table 3. Examples of interesting policies.

exceptions are re-thrown in the parent context after inspection from BreakApp running on the parent module.

If the specified module is already loaded, BreakApp simply retrieves the channel pointer and returns the previously-wrapped DAG.

The module system 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 waits for a response. The child-side wrapper deserializes arguments and calls the required method, sending results back through the channel. For asynchronous function calls, the parent module wrapper registers an event listener that invokes the provided continuation when results become available on the channel. In cases when something does not go as expected in the child's execution, its code will throw an exception. BreakApp code running on the child compartment will catch, serialize, and return it to the caller compartment, where the parent-side BreakApp code will deserialize and re-throw it.

Parent compartments naturally monitor the health (*i.e.*, crashed, not responding) of child compartments upon remote invocations, and take curative actions based on the exact status (*e.g.*, restart, kill, or spawn more compartments). This is helpful both in cases where the module within the compartment is launching a DoS attack as well as in cases where asynchronous execution has lead to exceptions (*e.g.*, access global variable *etc.*). Child compartments can use OS primitives (*e.g.*, `SIGHUP` on Linux) to be notified upon parent exit.

4 Overview of policies

Policies allow users to parametrize several aspects of the system's behavior (Table 3). The goal is to give them the flexibility to selectively disable capabilities the programming language gives modules. For example, if a module is not explicitly allowed to introspect or monkey-patch on core application structures or access global state, these capabilities can be disabled. Since policies express user insight, they can

also be used to fine-tune performance characteristics (*e.g.*, number of compartments and their types).

Policies can be set at the application-wide level or at the level of individual modules. Below is an example of how users express policies upon import:

```
var regex = require("minimatch",
      { type: require.types.PROCESS });
```

This specifies that the `minimatch` module should be loaded in its own, fresh process. OS-enforced isolation via processes provides better and potentially more costly isolation guarantees compared to a V8 runtime context.[2] However, these guarantees are probably worse (and certainly less expensive) than the ones provided by a virtual machine running on a different physical host.

Other examples of policies include `context`, a mapping object from bound variables to their values. Variables can point to values in different runtime contexts as a way to share state. Policies such as `instantiate` and `replicate` affect how many compartments to create per module. `onFail` is used to express possible actions when unexpected behavior is detected (*e.g.*, kill and restart compartment). We will see examples of uses in the next section.

Policy expressions are dynamic objects that can be generated during runtime. They can potentially change for each import — even between imports of the *same* module. This is a powerful feature, as different branches of a control flow statement might load the same module with a different policy. Composition options allow policies to freeze at the top level, trumping any other policy expression found in third-party modules.

Per module policy expressions are *fully* compatible with existing codebases. They are *backwards*-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 *forwards*-compatible with systems that *do* provide a BreakApp-enabled module system: as alluded to earlier, BreakApp will use the application-wide default configuration.

5 Discussion

We want to get a sense of the decomposition potential out in the wild, the space of possible security benefits, and the worst-case performance costs associated with applications.

5.1 Decomposition Potential

What are the modularity characteristics of JavaScript applications? In particular, is there a potential for compartmentalization? Table 4 describes some of the most popular JavaScript applications by four metrics: (i) only top-level modules, (ii) all modules within the dependency tree, (iii) lines of application code *not* part of a third-party module, and (iv) total lines of code in all of its third-party imports.

	Application	Direct Imports	Total Imports	App Code	Module Code
command	cash	15	84	15936	142098
	eslint	34	135	231907	171209
	yo	30	301	2005	27081
desktop	popcorn	46	765	103602	192082
	twitter	10	120	2951	419151
	atom	57	358	19879	223147
mobile	hackernews	5	871	2603	333975
	mattermost	17	521	11383	305664
	stockmarket	14	44	4473	406650
server	express	26	42	16906	10369
	ghost	62	981	96979	342676
	strider	64	659	32115	99051
utility	chalk	3	4	297	172
	natural	3	3	19741	5863
	winston	6	6	6229	2989
	averages	26	326	37800	178811

Table 4. Module usage in five categories of applications.

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; 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).

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). More fine-grained compartmentalization at the level of individual packages requires an order of magnitude more compartments (average: 326). Since there is a 1-1 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 a 43.09 average ratio of lines of code per file – much smaller than what we expected to see.[3]

All of the above show that there is a potential for compartmentalization, but also that the flexible granularity that policies offer is crucial.

[2] Google's V8 is a fast JIT compiler and runtime system for JavaScript. In V8 terms, a context is an execution environment that allows separate, unrelated executions in a single instance of V8. Similar mechanisms exist in other language runtimes [16, 48] or have been proposed at the OS level [24].

[3] As a point of comparison, Minix 3 [18], a modern microkernel that championed least-privilege separation, comes with userspace servers on the order of thousands of lines of code.

Compartments	InProc	FileSys	V8sbx	Proc	Function	Pipe	UDS	TCP
5	0.4ms	4.3ms	12.9ms	342.5ms	192.3GB/s	18.3GB/s	149.5MB/s	158.1MB/s
					2.5μs	1.3–1.4ms	17.8 – 73.8ms	17.7 – 36.6ms
50	0.4ms	30.2ms	76.6ms	3.2s	157.1GB/s	17.5GB/s	127.0MB/s	134MB/s
					3.18μs	11.6–13.2ms	244.5 – 536.6ms	210.3 – 566.8ms
500	0.5ms	136.4ms	524.7ms	35.2s	46.5GB/s	3.6GB/s	16.4MB/s	20.9MB/s
					10.74μs	154.3–160.3ms	3.71 – 11.95**s**	6.5 – 15.6**s**
5K	1.0ms	1.7s	7.8s	362.4s	—	—	—	—

Table 5. Compartmentalization costs. Left: module startup times. Right: throughput (and latency ranges) of boundary crossing.

5.2 Mitigation and Policies

Are there any third-party vulnerabilities out in the wild, and if yes, would BreakApp mitigate them? Table 2 contains a small set of distinct vulnerability classes along with several known instances found in the npm registry [41], caused by only a subset of the possible factors outlined at the end of Section 2. Revisiting the two example cases from Section 2 shows how a BreakApp-enhanced version of the blogging application would mitigate these attacks.

Example 1 Under BreakApp, the `ejs` module is placed into its own compartment: it does not have access to the application's global scope, it does not have access to the module cache, and — given a policy of restricting access within a new directory — it does not have access to the rest of the filesystem. If `ejs` attempts to access something it should not (*e.g.*, a global variable), an exception will be thrown and get caught by BreakApp.

Other configurations are also interesting: by spawning the database config module in its own compartment, we make sure no other module gets access to it beyond the parent module. The startup cost is negligible since there is only one extra compartment started. IPC costs are also amortized over long periods of time in which the database credentials are not needed (*i.e.*, the database config module is in the application's "control plane", not its "data plane").

Example 2 With BreakApp, there are at least two concurrent processes executing – one handling the matching of regular expressions (and communicating only with the main application) and the main application itself receiving requests. This is already a big win: search requests that do not contain regular expressions can still get served; it is only a subset of only the search functionality that remains paralyzed by the DoS attack.

But what happens when a *second* malicious regular expression arrives? Due to monitoring, BreakApp is at least aware that the previous request is taking more time than expected. By examining the input to the most recent RPC, it concludes that the new one is also problematic. Policies give users several options at this point: (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; (iv) discard or pushback based on history; or (v) by passing a function, implement their own action. From the examples above, as well as our own preliminary experiences, it seems that much will depend on identifying a set of "good-enough" defaults balancing performance and security. These default policies are a primary goal of our future evaluation.

5.3 Performance

What is the performance cost expected from using BreakApp? Table 5 highlights compartment startup costs (left) and the costs of boundary crossing (right) under various configurations. Large compartment counts were evaluated to anticipate the trend towards small modules (*e.g.*, maximum crossings of 500 when we did not witness more than 50). Experiments were executed on Andromeda [47] bundled with V8 v6.0 and LibUV v1.13 running on a Linux server with 512GB of memory and 160 cores at 2.27 GHz.

For the first experiment, we minimize the effects of module sizes by making modules return a single integer. Modules are loaded sequentially, since modules later in the program might be parametrized by values in earlier modules. *FileSys* 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. *V8sbx* creates a new V8 context for each module and selectively whitelists shared variables from the parent context. *Proc* uses OS processes to isolate compartments between each other, resulting in higher costs, illustrating one instance of the security-performance trade-off policies attempt to address.

InProc keeps all modules in memory and avoids all source code lookups but one (the first). It shows that the vanilla system *already* requires significantly more startup time (average for 5K modules: 0.3ms per module) compared to an optimized version (average for 5K modules: 0.2μs per module), without adding any meaningful security benefits.

For the second experiment, we process an in-memory stream of 0.5GB (`/dev/shm/`) using a linear pipeline of mostly-empty stages (they flush some timing metadata when they detect the end of a stream). Streaming starts only after all connections have been established. Loopback TCP streaming has

a higher throughput than domain socket streaming, but latencies are mixed. Experiments with fast userpace packet I/O libraries such as netmap [37] (not included here) and shared-memory pipes (included here) show much better latencies. This illustrates another reason why policies are important. Policies related to (i) the choice of compartment type and IPC primitive, or (ii) identification and expression of the critical path are both vital for high-performance applications.

5.4 *Can* we get good defaults?

There is indication that BreakApp can work in practice with better performance than our conservative worst-case estimates might suggest:

Shallow dependency trees Although the number of modules (hence, default compartments) is large due to fanout, the maximum number of boundary crossings is only related to the average depth of a tree – which seems much smaller (Applications in Table 4 have an average depth of only 6 levels).

Package deduplication Some of the *logical* dependencies are identical between different modules (*e.g.*, `lodash`, `underscore`). Although they show as distinct packages in the logical dependency tree, package managers usually deduplicate dependencies effectively flattening parts of the tree. For instance, `popcorn` went from 765 to 656 modules, resulting in 15% smaller tree.

Small working sets Not all package subtrees are in the critical path. In our example application, both the database configuration and the regex module have different usage patterns than the bulk of the application. They can still be protected by this technique, without being responsible for any hot-path overheads.

Parallel and lazy loading Dependency subtrees can take advantage of different loading strategies. Parallel startup is an obvious candidate: spawning 5K process compartments in parallel took 24s. Applications are naturally lazy in the way they load modules: we experimented with saturating disk bandwidth to the point where the 500 modules of the vanilla configuration took 3 seconds to load (instead of 136.4ms shown in Table 5 *FileSys*). Under such conditions, starting Ghost with 981 modules took under 0.4ms.

Other heuristics BreakApp can take advantage of interposition (by design) on boundary crossings and identify "hot boundaries" as candidates for merge – which can be done automatically after a prompt for security audit.

These insights, combined with performance-oriented policies (*e.g.*, group subtrees together in order reduce the costs, pick isolation guarantees *etc.*), suggest that a concrete instantiation of BreakApp could be made practical in *existing* systems. However, to take full advantage of the increasing trend towards small modules (by, say, isolating even the tiniest modules, re-instantiating on every load, and running multiple replicas), all while maintaining negligible throughput and latency degradation (Table 5) might require new insights in low-overhead OS compartmentalization mechanisms.

6 Related Work

Our concerns about large-scale reliance on loose supply chains are echoed by both academia [21, 27] and industry [7, 25, 43]. Several recent companies [32, 34, 43] 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 recently added support for locking dependencies between deployments [33]. 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.

There is a long history of alternative system structures with a focus on least privilege decomposition [40] and, more generally, separation of concerns [10] (*e.g.*, microkernels [18, 19, 23], capability systems [22, 42], and separation kernels [38]). Increasing security requirements brought decomposition to the foreground [35], culminating with systems such as Crowbar [4] and SOAAP [15] that assist programmers into decomposing applications into multiple compartments with reduced privileges. However, its wide adoption is still being impeded by lack of automation [27], the primary focus of this work. Many concrete sandboxing primitives can be used (*e.g.*, SELinux [26], AppArmor [3], Docker [28]) and we plan to experiment with some of them.

In the case of JavaScript specifically, much effort has gone into client-side compartmentalization (*e.g.*, execution isolation [29], object capabilities [30], sandboxing [1, 45] information flow control [44]). Our environment is very different from theirs — isolation primitives (iframes), origin (explicit sources), threat model (*i.e.*, no C/C++ modules; no valid access to "/etc/passwd"), compartments (few), and developer effort (manual annotations or rewrite).

More recently, microservice architectures – a style for building server applications as sets of loosely-coupled components [13, 31] – are often touted as enabling fine-grained, least-privilege decomposition inspired by the Unix philosophy. Even more so, lambda architectures [11, 17] 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 4. Moreover, (i) communication between services is request-response style and usually explicitly exposed to the application, (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, and our philosophy of enhancing security with minimal development effort.

7 Conclusion

This paper identifies and calls the community to harness a surprising potential in the way applications are built today: the use of many third-party modules, although risky, offers clear boundaries of trust. These boundaries can be used to automate parametrizable, compartmentalization-oriented transformations. Without any developer effort, a system can spawn modules into their own compartments and hide their boundaries from developers and users. Optional, flexible runtime policies let composers fine-tune security and performance trade-offs, essentially decoupling assumptions made during module development from requirements during the application runtime. *Our vision is that, with the synergy between linguistic and systemic levels, we can have applications with many, possibly dangerous, third-party packages be safer than their monolithic counterparts.*

The techniques presented integrate most naturally with interpreted languages such as JavaScript or Python. Their runtime environments 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. In compiled languages, these actions would be split between a compiler and a linker-loader. The compiler would be responsible for detecting compartment boundaries at import statements, marking object files as belonging to separate compartments, encoding policies for the linker-loader, and applying source transformations that will hide the boundaries. The runtime linker-loader would construct the compartments and channels during runtime. Since communication depends on policies, the linker would also load policy-specific shared libraries for handling compartment communication.

Acknowledgements We would like to thank Athur Azevedo de Amorim, Cătălin Hriţcu, Björn Knutsson, Yash Palkhiwala, Benjamin C. Pierce, and John Sonchack for their helpful feedback. We would also like to thank the anonymous reviewers for their comments and suggestions, and Maria G. Plani for coming up with the name BreakApp. 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] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2420950.2420952

[2] Slovakia's National Security Authority. 2017. skcsirt-sa-20170909-pypi. (Sep 2017). http://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/ Accessed: 2017-09-15.

[3] Mick Bauer. 2009. Paranoid penguin: AppArmor in Ubuntu 9. *Linux Journal* 2009, 185 (2009), 9. http://www.linuxjournal.com/magazine/paranoid-penguin-apparmor-ubuntu-9 Accessed: 2016-09-30.

[4] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322. http://dl.acm.org/citation.cfm?id=1387589.1387611

[5] Oscar Bolmsten. 2017. Looks like this npm package is stealing env variables on install. (Aug 2017). https://twitter.com/o_cee/status/892306836199800836 Accessed: 2017-08-11.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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 (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

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

[8] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11)*. ACM, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/2103799.2103805

[9] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1251353.1251356

[10] Edsger W Dijkstra. 1982. On the role of scientific thought. In *Selected writings on computing: a personal perspective*. Springer, 60–66.

[11] Marius Eriksen. 2013. Your Server As a Function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems (PLOS '13)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2525528.2525538

[12] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2429069.2429114

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

[14] Ghost. Ghost Publishing Platform. http://ghost.org/. (????). Accessed: 2017-01-01.

[15] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611

[16] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.* 410, 2-3 (Feb. 2009), 202–220. https://doi.org/10.1016/j.tcs.2008.09.019

[17] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[18] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: A Highly Reliable, Self-repairing Operating System. *SIGOPS Oper. Syst. Rev.* 40, 3 (July 2006), 80–89. https://doi.org/10.1145/1151374.1151391

[19] Michael J. Accetta, Robert Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Wayne Young. 1986. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer Technical Conference*. Usenix, 93–113. http://www.cs.ubc.ca/~norm/508/2009W1/mach_usenix86.pdf

[20] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. 2016. Diplomat: Using Delegations to Protect Community Repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 567–581. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/kuppusamy

[21] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[22] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA. http://www.cs.washington.edu/homes/levy/capabook/

[23] Jochen Liedtke, Kevin Elphinstone, Sebastian Schonberg, Hermann Hartig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. 1997. Achieved IPC performance (still the foundation for extensibility). In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE, 28–31.

[24] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[25] Jeremy Long. 2015. OWASP Dependency Check. (2015). https://www.owasp.org/index.php/OWASP_Dependency_Check Accessed: 2017-02-17.

[26] Peter Loscocco and Stephen Smalley. 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 29–42. http://dl.acm.org/citation.cfm?id=647054.715771

[27] Michael Maass. 2016. *A Theory and Tools for Applying Sandboxes Effectively*. Ph.D. Dissertation. CMU.

[28] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[29] James Mickens. 2014. Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains. In *2014 IEEE Symposium on Security and Privacy*. 261–275. https://doi.org/10.1109/SP.2014.24

[30] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. *Google, Inc., Tech. Rep* (2008).

[31] Sam Newman. 2015. *Building Microservices*. O'Reilly Media, Inc.

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

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

[34] Erlend Oftedal et al. 2016. RetireJS. (2016). http://retirejs.github.io/retire.js/ Accessed: 2017-05-18.

[35] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1251353.1251369

[36] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.

[37] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 101–112.

[38] J. M. Rushby. 1981. Design and Verification of Secure Systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/800216.806586

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

[40] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[41] Isaac Z. Schlueter et al. 2010. Node Package Manager. (2010). https://npmjs.com Accessed: 2017-02-17.

[42] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. ACM, New York, NY, USA, 170–185. https://doi.org/10.1145/319151.319163

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

[44] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 131–146. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan

[45] Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta. 2012. JavaScript in JavaScript (js.js): Sandboxing Third-Party Scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*. USENIX, Boston, MA, 95–100. https://www.usenix.org/conference/webapps12/technical-sessions/presentation/terrace

[46] Nikolai Philipp Tschacher. 2016. *Typosquatting in Programming Language Package Managers*. Bachelor Thesis. University of Hamburg.

[47] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. 2015. From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/vasilakis

[48] Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

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

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