Abstract—The software development community has begun to adopt the Event-Driven Architecture (EDA) to provide scalable web services. Though the Event-Driven Architecture can offer better scalability than the One Thread Per Client Architecture, its use exposes service providers to a Denial of Service attack that we call Event Handler Poisoning (EHP). With the rise in the use of the EDA may come a plague of EHP attacks threatening critical Internet services.

In this work we define EHP attacks, dividing them into two types, CPU-bound and I/O-bound. After examining EHP vulnerabilities in the popular Node.js EDA framework and open-source npm modules, we explore two representative EHP attacks in detail: Regular Expression Denial of Service (ReDoS) and I/O-based Denial of Service (IODoS).

Using our Constant Worst-Case Execution Time (C-WCET) Partitioning pattern, EDA-based services can become invulnerable to EHP attacks. Guided by this pattern, we present Node.cure, which defends Node.js applications against both ReDoS and IODoS. Our evaluation shows Node.cure to be effective. In assessing ReDoS, we extracted a novel corpus of 27,808 regular expressions from Node.js modules. Among these, Node.cure renders 51% of 4122 possible ReDoS vulnerabilities harmless without requiring application refactoring, including 96% of the 55 known ReDoS vulnerabilities. Meanwhile, Node.cure can detect and defeat File IODoS with a modest overhead.

I. INTRODUCTION

Web services are the lifeblood of the modern Internet. To handle the demands of millions of clients, service providers replicate their services across many servers. Since every server costs time and money, service providers want to maximize the number of clients each server can handle. Over the past decade, this goal has led the software community to seriously consider a paradigm shift in their software architecture — from the One Thread Per Client Architecture (OTPCA) used in Apache to the Event-Driven Architecture (EDA) championed by Node.js. We report that although using the EDA may increase the number of clients each server can handle, it also exposes the service to a new class of Denial of Service (DoS) attack unique to the EDA: Event Handler Poisoning (EHP).

The OTPCA and the EDA differ in the resources they dedicate to each client. In the OTPCA, each client is assigned a thread, and the overhead associated with each thread limits the number of concurrent clients the service can handle. In the EDA, a small pool of Event Handler threads is shared by all clients, and the application is structured to let each Event Handler accommodate multiple clients. The per-client overhead is thus smaller in the EDA, improving scalability.

Though researchers have investigated application- and language-level security issues that affect many services, we are the first to examine the security implications of using the EDA at a software architecture-level. In §III we describe a new Denial of Service attack that can be used against EDA-based services. Our Event Handler Poisoning attack identifies the most important limited resource in the EDA: the Event Handlers themselves.

The EDA’s scalability is also its Achilles’ heel. Multiplexing unrelated work onto the same thread reduces overhead, but it also moves the burden of time sharing out of the thread library or operating system and into the application itself. Where OTPCA-based services can rely on the preemptive multitasking promised by the underlying system to ensure that resources are shared fairly, using the EDA requires the service to enforce its own cooperative multitasking [48]. In other words, thread multiplexing destroys isolation. An EHP attack identifies a way to defeat the cooperative multitasking used by an EDA-based service, ensuring Event Handler(s) remain dedicated to the attacker rather than handling all clients fairly. We motivate this problem by identifying both CPU-bound and I/O-bound EHP attacks among previously-reported Node.js vulnerabilities (§III).

In §IV we present the C-WCET partitioning programming pattern, a general principle that EDA-based services can use to prevent EHP attacks. Because EHP attacks are possible when one request to an EDA-based server doesn’t cooperatively yield to other requests, our pattern is simple: every stage of request handling can take no more than constant time before it yields to other requests. In applying this principle, EDA-based servers extend input sanitization from the realm of data into the realm of the algorithms used to process it.

We evaluate the popular EDA-based Node.js ecosystem in light of this principle. We first report that the community at large does not seem to be aware of the problem (§V). We then identify both framework-level and application-level EHP vulnerabilities in the Node.js ecosystem. In case studies for ReDoS (§VI) and IODoS (§VII), our Node.cure prototype applies our C-WCET partitioning pattern to address the framework-level vulnerabilities we identified.

This paper is an extension of a workshop paper. There we introduced the idea of EHP attacks, argued that some previously-reported security vulnerabilities were actually EHP vulnerabilities, and gave a sketch of our antidote. Here we offer the complete package: a refined definition of EHP, examples of vulnerabilities in the wild, a general principle for defense, and effective framework-level defenses. Only in §V do we draw largely from our previously-published material.

In summary, this paper makes the following contributions:

1 The EDA paradigm is also known as the reactor pattern.

2 We omit the reference to meet the double-blind requirement.
1) We are the first to define Event Handler Poisoning (EHP) (§III), a DoS attack against the EDA. EHP attacks consume Event Handlers, a limited resource in the EDA. Our definition of EHP is general enough to accommodate all known EHP attacks.

2) We describe a general antidote for Event Handler Poisoning: C-WCET partitioning (§IV). Our C-WCET partitioning pattern gives developers precise guidance for defending against EHP attacks, replacing the heuristics used by the EDA community.

3) We perform case studies on two EHP vulnerabilities in the popular Node.js framework, showing the practicality of our C-WCET antidote even when applied to complex problems like ReDoS (§VI) and IODoS (§VII). In §VI we also extend the state of the art on ReDoS vulnerability detection.

II. BACKGROUND

In this section we contrast the EDA with the OTPCA (§II-A). We then explain our choice of EDA framework for study (§II-B), and finish by introducing formative work on Algorithmic Complexity attacks (§II-C).

A. Overview of the EDA

There are two main architectures for scalable web servers, the established One Thread Per Client Architecture (OTPCA) and the upstart Event-Driven Architecture (EDA). The OTPCA style, exemplified by the Apache Web Server\(^2\), each client is assigned a thread or a process, with a large number of concurrent clients. The OTPCA model isolates clients from one another, reducing the risk of interference. However, each additional client incurs memory and context switching overhead. In contrast, by multiplexing multiple client connections into a single thread and offering a small Worker Pool for asynchronous tasks, the EDA approach reduces a server’s per-client resource use. While the OTPCA and EDA architectures have the same expressive power \(^3\), in practice EDA-based servers can offer greater scaling \(^4\).

Although EDA approaches have been discussed and implemented in the academic and professional communities for decades (e.g. \([54], [5], [26], [20], [57], [46]\)), historically the EDA has only seen wide adoption in user interface settings like GUIs and web browsers. How things have changed. The EDA is increasingly relevant in systems programming today thanks largely to the explosion of interest in Node.js, an event-driven server-side JavaScript framework.

Several variations on the server-side EDA have been proposed: single-process event-driven (SPED), asymmetric multi-process event-driven (AMPED) \([43]\), and scalable event-driven (SEDA) \([57]\). The AMPED architecture is depicted in Figure 1, while in SPED there is no Worker Pool. In the SEDA architecture, applications are designed as a pipeline composed of stages, each of which resembles Figure 1.

From a software architectural perspective, Figure 1 illustrates the widely used AMPED EDA formulation, which is used by all mainstream general-purpose EDA frameworks today. In the AMPED EDA (hereafter simply “the EDA”) the operating system or a framework places events in a queue

---

3See https://httpd.apache.org/.

4See https://httpd.apache.org/

Vert.x\textsuperscript{9} (Java), Zotonic\textsuperscript{10} (Erlang), Twisted\textsuperscript{11} (Python), and Microsoft’s P\textsuperscript{12} [19]. These frameworks have been used to build a wide variety of industry and open-source services, at companies including LinkedIn\textsuperscript{13}, PayPal\textsuperscript{14}, eBay\textsuperscript{15}, and IBM\textsuperscript{15}, and for projects like Ubuntu One\textsuperscript{16}, Apple’s Calendar and Contacts Server\textsuperscript{17}, and IoT frameworks like CylonJs\textsuperscript{18}.

To capture the ongoing investment in each framework, we surveyed its GitHub\textsuperscript{19} repository for the number of contributors and forks (as a proxy for community interest), and commits (as a proxy for maturity). The results are shown in Table I. Using this data as a guide, we selected Node.js\textsuperscript{20} (JavaScript) for closer investigation, though EHP attacks apply to any EDA-based service.

Node.js is a server-side event-driven framework for JavaScript that uses the architecture shown in Figure 1. Born in 2009, Node.js now claims 3.5 million users\textsuperscript{21}, and as of August 2017 its open-source package ecosystem, npm, is the largest of any programming language or framework [1]. The Node.js Worker Pool is used internally for asynchronous encryption, compression, DNS queries, and file system I/O. In addition, user-defined code can be offloaded to the Worker Pool. The accepted best practice is to use the built-in cluster module to create a set of Node.js processes, one Event Loop per core, using message passing to communicate between each process.

C. Algorithmic Complexity Attacks

Our work is inspired by Algorithmic Complexity (AC) attacks, a form of Denial of Service attack [38], [15]. In an AC attack, attackers force victims to take longer than expected to perform an operation by incurring the worst-case complexity of the victim’s algorithms. Rather than overwhelming the victim with volume, the attacker turns the victim’s vulnerable algorithms against him. AC vulnerabilities can therefore be exploited to launch DoS attacks, when the attacker forces the victim to spend a long time processing his request at the expense of legitimate clients, or when the attacker increases the cost even of legitimate requests.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Contributors</th>
<th>Forks</th>
<th>Commits (last 6 mos.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js (JavaScript)</td>
<td>1436</td>
<td>7273</td>
<td>3790 (146)</td>
</tr>
<tr>
<td>libuv (C/C++)</td>
<td>209</td>
<td>1157</td>
<td>3900 (146)</td>
</tr>
<tr>
<td>EventMachine (Ruby)</td>
<td>121</td>
<td>554</td>
<td>1157 (16)</td>
</tr>
<tr>
<td>Vert.x (Java)</td>
<td>104</td>
<td>1177</td>
<td>2908 (448)</td>
</tr>
<tr>
<td>Zotonic (Erlang)</td>
<td>62</td>
<td>166</td>
<td>6753 (280)</td>
</tr>
<tr>
<td>Twisted (Python)</td>
<td>58</td>
<td>619</td>
<td>22161 (925)</td>
</tr>
<tr>
<td>P\textsuperscript{12}</td>
<td>11</td>
<td>18</td>
<td>1656 (296)</td>
</tr>
</tbody>
</table>

TABLE I. MATURETY OF EVENT-DRIVEN PROGRAMMING FRAMEWORKS MEASURED BY DEVELOPMENT ACTIVITY. DATA OBTAINED FROM GITHUB ON JULY 18, 2017.

Well-known examples of AC attacks include attacks on hash tables, whose worst-case $O(N)$ behavior can be triggered by attackers [15], and Regular expression Denial of Service (ReDoS) attacks (§VI), which exploit the worst-case exponential-time behavior typical of regular expression (regex) engines [14], [47], [53], [31], [45].

Our work is not simply the application of AC attacks to the EDA. The first difference is a matter of perspective: EHP attacks target the software architecture used by the service, while AC attacks focus on the algorithms the service employs. The second difference is one of definition: AC attacks must rely on CPU-bound activities to achieve DoS, while EHP attacks may freely use either CPU-bound or I/O-bound activities, so long as they can poison an Event Handler.

III. EVENT HANDLER POISONING ATTACKS

In this section we provide our threat model (§III-A), define Event Handler Poisoning (EHP) attacks more formally (§III-B), and show their presence in Node.js’s npm ecosystem with a novel analysis of previously-reported security vulnerabilities (§III-D). To help illustrate the problem, we present simple EHP attacks using ReDoS and IODoS (§III-C). Lastly, we perform a manual survey of npm modules to determine how aware the community seems to be of EHP attacks (§III-D).

A. Threat Model

Our threat model is straightforward: the attacker crafts evil input that triggers an expensive operation, and convinces the EDA-based victim to feed it to a Vulnerable API. An ideal EHP is asymmetric, such that a small attacker input can trigger a large amount of work on the part of the poisoned event handler. If so, the attacker can perform a small amount of work and poison the victim. Depending on the degree of the expensive operation, the attacker can achieve DoS outright, or can repeatedly apply poison in a Slow DoS attack [8].

This model is reasonable because our victim is an EDA-based server; servers exist solely to process client input using APIs. If the server uses a vulnerable API, and if it invokes it with unsanitized client input, the server can be poisoned. We will show examples of vulnerable APIs in both EDA-based applications and frameworks, focusing on Node.js.

B. What is an EHP attack?

Server-side applications implemented using the EDA are vulnerable to a unique type of denial of service attack: Event Handler Poisoning. EHP vulnerabilities exist when applications incorrectly make use of the EDA.

Some definitions. Before we can define EHP, we must introduce a few definitions. First, recall the standard EDA formulation illustrated in Figure 1. Each request is partitioned into a series of one or more events (e.g. “new connection”, “file read request”, “response ready”) to be handled by the corresponding event callbacks. These event chains (or lifelines [3]) always begin with one initial event often triggered by a client request. An event chain maps to an asynchronous call stack whose root function is its initial event’s callback.

We define the synchronous complexity of an event chain as the greatest complexity among all event callbacks in the
An EHP attack exploits this architecture by poisoning one or more Event Handlers with evil input. An attacker first identifies a vulnerable event chain, an event chain with a large worst-case synchronous complexity. He then submits a request that triggers this worst-case complexity. The resulting event chain’s large synchronous complexity causes the affected Event Handlers to block, perhaps indefinitely, while handling the expensive event. While an Event Handler is blocked, it will not continue to process other events, and the clients who originated those events will starve. Because it exploits a fundamental property of the EDA, namely the need to correctly implement fair cooperative multitasking, an EHP attack is simple to understand. This makes it no less deadly.

As a convenient taxonomy, we divide EHP attacks into CPU-bound attacks, whose evil input triggers computationally-intensive activity (e.g. via AC attacks), and into I/O-bound attacks, whose evil input triggers I/O-intensive activity. We explore a CPU-bound attack in §VI and an I/O-bound attack in §VII.

An EHP attack can be carried out against either type of Event Handler, viz. the Event Loop and the Workers in the Worker Pool, though the two types respond differently when poisoned. If the Event Loop is poisoned, neither pending nor future requests will get a response. On the other hand, for each poisoned Worker the responsiveness of the Worker Pool will degrade, until every Worker is poisoned and all pending and future requests whose event chains lead to the Worker Pool will receive no responses. Thus, an attacker’s aim is to poison either the Event Loop or the entire Worker Pool.

You might be surprised that we name the Worker Pool as a potential target for EHP attacks. Remember, however, that in the EDA the Worker Pool is deliberately quite small, as part of the EDA’s low-overhead philosophy. For example, in a Node.js process the Worker Pool defaults to 4 Workers, with a maximum of 128 Workers. Though a small Worker Pool is in keeping with the spirit of the EDA, its size makes it a realistic target. In contrast, though typical OTPCA-based servers also maintain a pool of threads (Workers) to bound the total number of concurrent connections, this pool is comparatively enormous. Apache’s httpd server has a default connection pool size of 256-400, with a maximum around 20,000. Due to the size of an OTPCA server’s “Worker Pool”, attempts to poison it would presumably be detected by an IDS or addressed through rate limiting before they poisoned enough of the pool to have an effect. By comparison, the small number of requests needed to poison a vulnerable EDA server’s Worker Pool is too few to draw attention from network-level defenses, placing the burden of defense on the application developer or the runtime system to address it.

We believe EHP attacks on the EDA have not previously been explored because, as noted in §II-A, the EDA has only recently seen popular adoption by the server-side community. On the client side the need to consider EHP attacks is limited, as a misbehaving user will hurt only himself. On the server side, the stakes are higher, especially as the EDA is adopted in safety-sensitive settings like IoT controllers and robotics.

C. Two simple EHP attacks

We prepared an example to show what an EHP attack can do. Our simple file server is shown in Figure 3, and has two EHP vulnerabilities. First, it sanitizes input with a regexp vulnerable to ReDoS. On evil input the regexp evaluation will take a long time, poisoning the Event Loop in a CPU-bound EHP attack. Second, its sanitization is inadequate: it will read arbitrary files requested by the user, exposing it to directory traversal attacks. In the EDA, directory traversal attacks can be parlayed into I/O-bound EHP attacks, IODoS. See §VI and §VII for a deeper exploration of these EHP vulnerabilities.

In Figure 4 you can see the baseline server throughput (circles) and the throughput while undergoing two attacks (IODoS in squares, ReDoS in triangles), averaged over five runs. In each condition we configured a Worker Pool of size 4, and used the ab benchmark to make 2500 legitimate requests per second divided among 80 clients. We measured the instantaneous throughput of the server at 200 ms intervals.

In the IODoS case, at seconds 1, 2, 3, and 4 an attacker submits one read request to a Slow File. Each malicious request poisons one of the four Workers. After three Workers

**Fig. 2.** These are two Python-esque functions that map a function \( f \) to each element of a list of length \( N \). Suppose that applying function \( f \) to any element of \( \text{list} \) has complexity \( O(f) \). Then function \( \text{map}_O \_Nf \), which synchronously applies \( f \) to each element in \( \text{list} \), has synchronous complexity \( N \times O(f) \). Function \( \text{map}_O \_f \) uses an async-await idiom to start an event chain of \( N \) events, each of which synchronously applies \( f \) to one element. Thus, its synchronous complexity is \( O(f) \).

```python
1 def map_O_Nf (list, f):  # f is synchronous
2     for e in list:
3         f(e)
4
5 async def map_O_f (list, f):  # f is asynchronous
6     for e in list:
7         await f(e)
```

See e.g. the values for ServerLimit and MaxRequest Workers in the Apache httpd documentation, available at https://httpd.apache.org/docs/2.4/mod/.

23See https://httpd.apache.org/docs/2.4/programs/ab.html.

24We simulated a Slow File, e.g. a file accessed over NFS, using nbd and trickle.
we performed a detailed analysis only of Snyk.io’s database. In this section we summarize the vulnerabilities in the Snyk.io database and analyze them through the lens of EHP attacks.

First, we divided the vulnerabilities by category. Each raw Snyk.io database entry includes a title and description, and some have additional material like CWE labels and links to GitHub patches. Since the titles indicate the high-level exploit, e.g. “Directory Traversal in servergmf...”, we sorted the npm vulnerabilities into 17 common categories and one Other category for the remaining 40. Figure 5 shows the distribution, absorbing categories with fewer than 10 vulnerabilities into Other. A high-level CWE number is given next to each class.

Then, we studied each vulnerability to see if it could be employed in an EHP attack. The dark bars in Figure 5 indicate the 177 (25%) vulnerabilities that we found could be used to launch an EHP under our threat model. The 145 EHP-relevant Directory Traversal vulnerabilities are exploitable because they allow arbitrary file reads, which can poison the Worker Pool. The 32 EHP-relevant Denial of Service vulnerabilities poison the Event Loop; 28 are ReDoS, 3 lead to infinite loops, and 1 allows an attacker to trigger worst-case algorithm behavior in input processing. We also note that the 65 Command Injection vulnerabilities could of course be employed for EHP attacks, and that the 9 Arbitrary File Write vulnerabilities could lead to EHP under a stronger threat model.

As a final note, since npm has approximately 500,000 modules, we were surprised that only 713 vulnerabilities have been reported. Caution dictates that many more security vulnerabilities remain hidden.

IV. AN ANTIDOTE FOR EVENT HANDLER POISONING

As defined in III, an EHP attack exploits an event chain with large worst-case synchronous complexity. An EDA-based service can defend against EHP attacks in several ways.

Reduce functionality. The service can remove the vulnerable functionality, but ad absurdum, “No service, no vulnerability”. This is not a general solution.

Input sanitization. The service can use traditional input sanitization, perhaps applying regexps to confirm that the input

---

```javascript
1 const _HTTP = require('http'),
2 _URL = require('url'),
3 _FS = require('fs');
4 const cb = (req, resp) => {
5   let url = _URL.parse(req.url, true);
6   let f = url.query.fileToRead;
7   if (f.match(/(a+){40}$/)) // ReDoS
8       resp.end('Invalid file');
9   else {
10      _FS.readFile(f, (err, d) => { // Dir. trav.
11         resp.end('Finished read');
12       });
13   }
14 }
15 const srv = _HTTP.createServer(cb).listen(3000);
```

---

25 See https://nodesecurity.io/
26 See https://snyk.io/
27 Available at https://snyk.io/vuln/feed.xml.
28 See https://www.npmjs.com/ or [1].
is not harmful. This kind of piecemeal defense is infeasible, not least because §III showed that regexps themselves are a potential source of EHP attack. *Quis custodiet ipsos custodes?*

**Timeouts.** More reasonably, the service can employ timeouts to cap the maximum cost of an event or an event chain. While a timeout ensures that the assigned Event Handler will not block indefinitely, timeouts come with two caveats. First, timeouts must be part of the existing framework and language specification. While this is the case for some languages, others, like JavaScript, do not define them. Introducing timeouts into frameworks like Node.js would require significant refactoring of both frameworks and applications. Second, the length of a timeout is a Goldilocks problem; it must be long enough to accommodate valid requests, but short enough to render EHP attacks ineffective. Conservatively choosing larger timeout values will still leave a service vulnerable to EHP-based Slow DoS attacks [8], analogous to Crosby and Wallach’s attacks on IDS systems [15].

**C-WCET Partitioning.** C-WCET Partitioning, our proposed defense, strikes at the heart of an EHP attack. If the service has no event chain with large worst-case synchronous complexity, no EHP attack can be made. We argue, then, that EDA-based services should bound the worst-case synchronous complexity of their event chains. Any event chain whose synchronous complexity is a function of its input could in principle be exploited, so we propose that *EDA-based services should use event chains with constant synchronous complexity, composed of events with (small) Constant Worst-Case Execution Time (C-WCET)*. A C-WCET-partitioned algorithm can block neither the Event Loop nor the Worker Pool, and thus a service composed of such algorithms is invulnerable to EHP attacks. C-WCET partitioning is essentially the logical extreme of cooperative multitasking, and interprets EDA systems as real-time systems with a need for strict forward progress guarantees [58].

The obvious shortcoming of C-WCET partitioning is that, like timeouts, it requires refactoring EDA-based services to use only C-WCET-partitioned algorithms. However, we think that failing to use C-WCET partitioning is insecure. Developers should view C-WCET partitioning as a form of input sanitization. Given the responsiveness requirements in the EDA, input sanitization must extend from the realm of data into the realm of the algorithms used to process it. When using the EDA, C-WCET partitioning is as necessary as checking user input for directory traversal attacks or code injection.

With our C-WCET defense in mind, we can now evaluate the common work patterns used in EDA-based services. As shown in Figure 6, work can be performed on the Event Loop (a), in the Worker Pool (b-d), or by an external service (e). When a service does not use a C-WCET-partitioned pattern, it is vulnerable to EHP, whether on the Event Loop (a) or on the Worker Pool (b, c). Note that in (c) the work is partitioned, but as one of the pieces is not constant-time, it remains vulnerable to EHP attack. In (d), an expensive task is performed on the Worker Pool in a C-WCET-partitioned manner, always cooperatively yielding to other events. In (e), an expensive task is offloaded from the Event Loop to an external service in constant time, and a response is eventually delivered. Pattern (e) is commonly used to communicate with components whose APIs are network-based.

**Community Awareness.** The research community has discussed the problem of performing work in an EDA-based environment, but only on the client side. There the concern is avoiding the performance bugs [36] that result from blocking the UI thread (Event Loop). To this end, Lin et al. have proposed an automatic refactoring to offload expensive tasks from the Event Loop to the Worker Pool [35] (Figure 6 (a) → (b)). This approach is sound on a client-side system like Android, but is not applicable to the server-side EDA. From a language perspective, Okur et al. offer tools to ease the migration from older callback-based asynchronous idioms to the modern async/await style in the .NET framework [41]. We are not aware of research work discussing the use of the general EDA on the server side and the resulting EHP vulnerability.

The server-side EDA community is aware of the risk of DoS due to EHP on the Event Loop. A common rule of thumb is “Don’t block the Event Loop”, advised by many tutorials as well as recent books about EDA programming for Node.js [55], [9]. Wandschneider suggests worst-case linear-time partitioning on the Event Loop [55], still unsafe if the $O(N)$ operation is expensive. Casciaro advises developers to partition any computation on the Event Loop, and to offload computationally expensive tasks to the Worker Pool [9], but he doesn’t differentiate between unsafe pattern (c) and safe pattern (d) (Figure 6).

**V. CURRENT PRACTICES IN NODE.JS**

Having explored EHP attacks (§III) and the various possibilities for defense (§IV), a natural question is:

**RQ1** How does the Node.js community handle expensive tasks?

To answer this question, we looked first at the Node.js framework itself, and then performed a small survey of popular npm modules used by Node.js applications. We understand that

---

29Not shown is an approach similar to (d), but with the work performed on the Event Loop. The Event Loop should generally be reserved for orchestration tasks to maximize the number of concurrent clients.

30For example, the npm module for Redis uses this pattern. See https://www.npmjs.com/package/redis.
the limited scope of this section raises more questions than it answers, but we hope it whets our readers’ appetites for further investigation. In summary, we found that: (1) while the Node.js framework is implemented relatively well with regard to EHP attacks, it still has vulnerabilities, and (2) the npm community is more cavalier about the EHP problem in the EDA.

The Node.js Framework. The core components of the Node.js framework are fairly limited, comprising support for JavaScript via Google’s V8 JavaScript engine, modules for computational tasks like encryption and compression, and modules for I/O tasks like network operations and file system access. The V8 engine is high performance, minimizing the risk that standard JavaScript operations (e.g. array or object accesses) might expose an application to EHP. The most notable exception is regexps, which we explore in more detail in §VI. Node.js’s built-in computationally-intensive tasks — encryption and compression — are both implemented using a C-WCET design (Figure 6 (d)) partitioned at the block granularity. The Node.js APIs for I/O tasks are more varied, including, with reference to Figure 6, (e)-style (e.g. the HTTP APIs), (c)-style (e.g. the asynchronous file stream APIs), (b)-style (e.g. the asynchronous DNS lookup API), and (a)-style (e.g. the synchronous file APIs).

A lingering question is whether the framework documentation is effective with an eye to EHP attacks, and we make two observations in this regard. First, nowhere in the framework documentation is there a careful discussion of the risks of high-complexity operations or of appropriate work patterns, and the framework APIs do not document their computational complexities. The framework developers demand that their users simply trust them. For example, the implementation must be studied to determine whether the asynchronous compression and encryption APIs follow patterns (b), (c), (d), or (e). Second, we found warnings about the risk of Worker Pool poisoning by long-running operations in only one place31, although APIs in the file system are also risky (§VII).

npm modules. In December 2016 we examined 80 npm modules in detail to determine how they handle potentially computationally expensive tasks, manually inspecting both documentation and implementation. Using npm’s “Best overall” sorting scheme, we examined the first 20 (unique) modules returned by npm searches for string, string manipulation, math, and algorithm, in hopes of striking a balance between popularity and potential complexity. In all, these modules were downloaded over 200 million times that month. Table II summarizes the APIs these modules offered.

We found that many of these modules perform non-partitioned expensive operations, leaving callers vulnerable to EHP attack. Although the execution time of an API is critical to preventing EHP, the documentation for the majority of the APIs did not state their running times. Nearly all of the hundreds of APIs we examined executed synchronously on the Event Loop; two were asynchronous, and one of these synchronously executed an exponential-time algorithm before yielding control. During our analysis of the source code we identified algorithms with complexities including $O(1)^{32}$, $O(n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$.

In summary, we found that these modules offer almost no asynchronous or C-WCET-partitioned APIs. This is in keeping with a previous study on the use of functions that examined call-sites in server-side JavaScript [25]. While Node.js application developers could offload these synchronous calls to the Worker Pool, doing so merely moves the potential poison from the Event Loop to the Worker Pool (§III).

VI. Case Study: ReDoS in Node.js

Though some EHP vulnerabilities are the responsibility of the application developer, in other cases the blame lies with the framework. Using Node.js as an example, we identify two cases where Node.js’s infelicitous implementation exposes applications to EHP attacks, and in our Node.cure prototype we apply the C-WCET partitioning pattern to these cases. In this section we discuss ReDoS, and in §VII we discuss IODoS. Our prototype defenses explore how best to perform CPU-bound and I/O-bound tasks in EDA-based services.

ReDoS attacks. Regular expression Denial of Service (ReDoS) is a class of AC attack that has gained attention in recent years [47], [53], [31], [45]. Developers often use regexps for input sanitization. Now, many popular regexp engines (Perl, Python, V8 JavaScript, etc.) have worst-case exponential-time performance. In consequence, a poor choice of regexp can expose an application to ReDoS from evil input that triggers this worst-case performance.

This worst-case exponential-time performance is necessary, but only sometimes. Production-grade “Regular Expression” engines are misnamed: they support backreferences32, a Context-Free Language extension that must in the worst case require exponential time. This worst-case behavior is also known as “catastrophic backtracking” because the implementations typically consume input greedily and on mismatch will “backtrack” to a previous point in the input string. However, in many cases this backtracking is not strictly necessary but is rather used to simplify implementations. For example, consider the regexps \/(a+)\15/ and \/(a+)\+$/, which match much the same thing. The former has worst-case exponential-time because the engine must remember the capturing group it chose and may backtrack on mismatch, while the latter can be worst-case linear-time because it can be defined by a simple DFA. However, the latter is still vulnerable (exponential-time performance) in many regexp engines, including those of Python and Node.js, because these engines use backtracking when not strictly necessary.

In the EDA, ReDoS attacks are a form of EHP attack. If the regexp engine evaluates regexps synchronously (as in Python, Ruby, Perl, JavaScript, etc.), ReDoS will poison the Event

<table>
<thead>
<tr>
<th>Search string</th>
<th># async APIs</th>
<th># C-WCET</th>
<th># Offloading</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
| string
manipulation| 1           | 0        | 1           |
| math          | 0           | 0        | 0           |
| algorithm     | 1           | 0        | 1           |

Table II. This table shows the use of asynchronous APIs and C-WCET partitioning in a sample of npm modules. Values are for all 20 modules in each category. # async is the number of async APIs. # C-WCET is the number of APIs that take a C-WCET approach. # Offloading is the number of APIs that offload tasks to a Worker Pool.

31The documentation for the DNS lookup API warns of this risk.

32In fairness, a synchronous $O(1)$ algorithm is C-WCET with one partition.

33A backreference \$i will match the substring from the \$i\text{th} capturing group.
Loop (Figure 6 (a))\textsuperscript{34}. As regexps are widely used, ReDoS is an attractive avenue of EHP attack on EDA-based services.

Here we consider ReDoS attacks in the Node.js community. This case study is guided by three research questions:

\textbf{RQ2} Can a developer identify ReDoS vulnerabilities?
\textbf{RQ3} Do npm modules have ReDoS vulnerabilities?
\textbf{RQ4} Can we defend against ReDoS?

\section*{A. RA2: Identifying ReDoS vulnerabilities is hard}

A seemingly straightforward defense against ReDoS is to avoid the use of vulnerable regexps. Indeed, there are several \textit{ReDoS Oracles} that accept a regexp and determine whether or not it is vulnerable. In this section we demonstrate that the existing ReDoS Oracles are neither accurate nor adequate.

\textbf{ReDoS Oracles.} We used an Internet search engine to identify ReDoS oracles, using queries like “detect vulnerable regexp”. We found two open-source oracles: \texttt{safe-regex}\textsuperscript{35} and \texttt{rxxr2} \textsuperscript{36}. \texttt{safe-regex} simply labels as vulnerable all regexps with star height \textsuperscript{28} greater than one (i.e. nested quantifiers), while \texttt{rxxr2} performs a more complex analysis based on an idealized backtracking regexp engine. To the best of our knowledge, ours is the first comparison of these oracles.

An ReDoS Oracle should recommend the evil input that exploits the vulnerable regexp. This evil input is composed of three parts: the \textit{prefix}, the \textit{pump}, and the \textit{suffix}. Since backtracking behavior is induced by requiring the regexp engine to try exponentially many paths through the input before declaring a mismatch, the attacker uses the prefix to reach the vulnerable point in the regexp, repeats (or “pumps”) the pump string to repeatedly double the size of the tree explored by the engine, and adds the suffix to cause a mismatch and trigger the backtracking. Among our oracles, \texttt{rxxr2} recommends evil input, while \texttt{safe-regex} does not.

\textbf{Evaluation.} We used our Snyk.io npm vulnerability study (§III-D) to obtain ground truth on which to evaluate these oracles. Since the Snyk.io vulnerability reports do not specify the vulnerable regexp(s), we extracted many regexps from the 19 npm modules reported by Snyk.io to have had ReDoS vulnerabilities as of February 12, 2017, and manually determined whether each regexp was vulnerable. We found that 20 of the 43 regexps in this corpus were vulnerable. We then evaluated the regexps using \texttt{safe-regex} and \texttt{rxxr2}. Their performance is given in Table III. We defined a regexp as vulnerable if Node.js took more than 500 ms to evaluate the regexp on an evil input that was “pumped” at most 20,000 times. This approach may have false negatives, as it will filter out any vulnerabilities that manifest on (absurdly) long input. When \texttt{rxxr2} reported a vulnerability, we used its recommendation as evil input. When \texttt{safe-regex} reported a vulnerability that \texttt{rxxr2} did not, we derived evil input manually. As noted in \cite{45}, automatically deriving evil input in general is difficult, and we did not attempt to add this feature to \texttt{safe-regex}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Oracle & Safe & Vuln. & Correct & False + ves & False - ves \\
\hline
\texttt{rxxr2} & 28 & 15 & 38 & 0 & 5 \\
\texttt{safe-regex} & 14 & 29 & 25 & 12 & 37 \\
\hline
\end{tabular}
\caption{Decisions and Accuracy of the \texttt{rxxr2} and (patched) \texttt{safe-regex} ReDoS Oracles. A “positive” decision marks a regexp vulnerable. The oracles agreed on 23 out of 43 decisions.}
\end{table}

\section*{Analysis.} We found the oracles untrustworthy, both alone and when used in combination. Here is a detailed assessment of their performance.

1) \texttt{rxxr2} reported no false positives. Each of its suggested evil inputs led to ReDoS.
2) \texttt{safe-regex} reported false positives. These are regexps that could in principle be vulnerable but which are not in Node.js. In our corpus, these false positives came in two forms: apparently non-dangerous quantifiers like “a?” and small specific quantifiers like “a[2]”.
3) \texttt{rxxr2} is unsound in practice, and \texttt{safe-regex} reported true positives that \texttt{rxxr2} did not. It is not clear whether \texttt{rxxr2}’s unsoundness stems from its design or its implementation.
4) \texttt{safe-regex} is fundamentally unsound, because catastrophic backtracking can be induced without nested quantifiers. The true positives found by \texttt{rxxr2} but not \texttt{safe-regex} took two forms: nested quantifiers with an OR’d inner expression (e.g. \texttt{/ (X+|Y)+$/}), and quantified expressions with overlapping OR’d classes (e.g. \texttt{/ (X|Y)+$/}). This latter kind is vulnerable because on input of the form “X...X!”, a backtracking engine will try every possible division of the X’s between the first and second OR’d classes.
5) \texttt{rxxr2} does not identify a similar form of vulnerable regexp: adjacent quantified expressions with overlapping contents, like this one: \texttt{/\d+\d+$/}. This is outside the scope of \texttt{safe-regex}, though curiously its test suite includes regexps of this form in its list of “safe” regexps.
6) Ironically, and contrary to the claims of the authors \cite{45}, \texttt{rxxr2} does not seem to consider backreferences vulnerable (e.g. the simplest such vulnerable regexp \texttt{/ (a+)\1$/}). These are outside the scope of \texttt{safe-regex}.

\section*{B. RA3: There are ReDoS vulnerabilities in npm modules}

In this section we describe how we found at least 55 ReDoS vulnerabilities among the 27,808 used in the npm modules we studied. Our methodology was straightforward: identify popular, well-maintained npm modules, extract their regexps, classify them using ReDoS oracles, and try evil input.

\textbf{Identifying popular, well-maintained npm modules.} To identify novel ReDoS vulnerabilities, we first had to identify regexps used in real, well-maintained software, which we selected from the Node.js package ecosystem, npm. We searched npm for popular (many downloads), well-maintained (many commits) packages, using these steps:

1) We cloned the npm registry on May 3, 2017. This registry contains the metadata for each of the 447,362 modules defined in the npm ecosystem.
2) We filtered the registry for modules whose code is hosted on GitHub, which has an easy-to-query API from which we

\textsuperscript{34}C#’s regexp engine supports timeouts as of .NET 4.5. It is the only major regexp engine to do so, and applications still must opt in.

\textsuperscript{35}See https://github.com/substrack/safe-regex.

\textsuperscript{36}We are aware of one other potential ReDoS oracle, Microsoft’s SDL Regex Fuzzer \textsuperscript{52}. However, this tool is no longer available.

\textsuperscript{37}These false negatives from \texttt{safe-regex} were due to a bug that we have patched. See our pull request at BLINDED.
could extract commit numbers. From this set of 348,691 modules we excluded 78,890 modules which had no monthly downloads or for which we did not identify at least one commit (e.g. a missing GitHub repository).

3) For the remaining 269,801 modules, we obtained popularity data (monthly downloads) from npm and maintenance data (commits) from GitHub. We only counted the commits made by the project’s top 30 contributors due to GitHub’s rate limiting policy.

4) We defined as popular and well-maintained the 1,591 modules (0.5%) that fell at least two standard deviations above the arithmetic mean in each dimension (i.e. at least 270 commits and 3500 downloads per month). The majority (250,164) were below this standard in both dimensions.

**Extracting regexps.** Having obtained a dataset of popular, well-maintained npm modules, our next step was to extract the regexps they used. To this end, we instrumented Node.js LTS v6.10.3\(^{39}\) to emit regexps as they were declared. We favor this dynamic approach over the static analysis used by [12]: a dynamic approach ensures that only regexps in active use are considered, and can handle dynamically defined regexps.

We then installed each module and ran its test suite. Though we don’t expect the test suites to offer complete code coverage, regexps are typically used to sanitize input, and thus we expect a reasonable test suite to exercise most if not all of the regexps used by the module.

We were able to successfully run the test suites of 1,182 of our 1,591 chosen modules\(^{40}\), capturing 2,282,330 regexp declarations, which we reduced to a corpus of 27,808 unique regexps\(^{41}\). The large number of repeated regexps was likely due to our use of dynamic analysis, since different tests will traverse the same input sanitization code.

**Results and conclusions.** We then applied our ReDoS oracles to these 27,808 regexps, with two tweaks. First, as noted in §VI-A, we used our patched version of safe-regex. And second, before doing so, we addressed the shortcoming described there in #6: neither safe-regex nor rxxr2 considers exponential-time regexp features, resulting in false negatives. To eliminate these false negatives, we developed our own simple but novel oracle, has-exp-features. has-exp-features identifies regexps that use exponential-time features and labels them vulnerable. has-exp-features flags backreferences as well as lookahead assertions, which need not be exponential-time but are especially difficult to implement without backtracking [13]. has-exp-features thus has no false negatives but may have false positives.

Armed with three ReDoS oracles, two existing and one novel, we classified the wild regexps. Figure 7 illustrates the varying opinions the oracles gave on the 4,122 regexps (15%) reported “vulnerable” by at least one oracle. rxxr2 (RX) was the most conservative, while safe-regex (SR) and has-exp-features (HE) flagged similar quantities of expressions. The small overlap between safe-regex and has-exp-features indicates that the regexps in our corpus are not “too complicated” – few expressions used both nested quantifiers and backreferences or lookahead.

![Fig. 7. This figure shows the overlap between the 4,122 regexps labeled vulnerable by at least one ReDoS oracle on our corpus of 27,808 wild regexps. RX denotes rxxr2, SR denotes safe-regex, and HE denotes has-exp-features. We have dynamically demonstrated ReDoS attacks on 55 of these regexps using the evil input recommended by rxxr2.](image)

We used rxxr2’s recommendations for evil input to identify 55 regexps from the npm corpus as truly vulnerable. As rxxr2’s recommendation format is ill-defined, we were unable to automatically reach conclusions on the remaining 101 regexps it flagged as vulnerable. Since neither safe-regex nor has-exp-features recommends evil input, we were unable to validate these dynamically. The inaccuracies of the oracles (§VI-A), however, suggests that there are likely truly vulnerable regexps in each area of the figure. The development of improved ReDoS oracles remains an area for future work.

Though we identified many possibly- and truly-vulnerable regexps, we did not differentiate between exploitable and non-exploitable vulnerabilities, for two reasons. First, our experience is that regexps tend to be re-used, so today’s non-exploitable vulnerability could be tomorrow’s attack vector. Second, and more critically, we found that relatively few regexps from our corpus (7%, about half of the vulnerable regexps) used exponential-time features. The remaining 93% represent an opportunity\(^{42}\). Some are vulnerable due to the regexp engine implementation, whether or not our oracles detected them, but none need to be. In the next section we study how to make them safe.

**C. RA4: Many ReDoS attacks can be prevented**

As noted in §VI-B, half of the vulnerabilities reported by our ReDoS oracles are preventable. Though synchronous evaluation of regexps that use exponential-time features is unavoidably vulnerable, the remainder would be safe were they evaluated using a linear-time regexp engine instead of Node.js’s exponential-time engine, Irregexp\(^{43}\). The opportunity is illustrated in Figure 7: the potentially-vulnerable expressions not flagged by has-exp-features (52%) could be made safe. We have implemented a hybrid regexp engine for Node.js to protect these vulnerabilities.

**Node.cure Design.** In our Node.cure prototype we modified the implementation of regexps in the underlying V8

\(^{38}\)npm is the largest package ecosystem [1], [59]. Is size everything?

\(^{39}\)We applied our changes to Node.js commit a08fd4f5 (5 May 2017)

\(^{40}\)Of the remainder, some test suites would not run, and others timed out under our five minute cap.

\(^{41}\)Uniqueness was determined by the regexp pattern alone, consolidating regexps that used different flags.

\(^{42}\)The incidence of exponential-time features in our corpus is comparable to that reported by Chapman and Stolee [12] in their analysis of regexps used in Python projects. They reported backreferences and lookahead assertions in 0.4% and 1.4% of 13,711 regexps, respectively.

\(^{43}\)Regexp processing is part of the V8 JavaScript engine. Actually, V8’s regexp engine is already a simple hybrid. It uses Atom to service simple regexps that can be handled with strstr, and Irregexp for the rest.
JavaScript engine. For the linear-time engine we used RE2\textsuperscript{44}, which supports perl-compatible regexps (PCRE) and is the most prominent such linear-time engine.

We began with a conservative design that referred all RE2-supported evaluations to RE2, but found that RE2 is significantly slower than Irregexp on safe regexps. Defaulting to RE2 was 1.2x and 1.6x slower than using Irregexp on the Li\textsuperscript{45} and V8 Octane\textsuperscript{46} regexp benchmarks, respectively. To improve Node.cure’s steady-state regexp evaluation performance, we incorporated our three ReDoS oracles to direct only vulnerable regexps to RE2, favoring Irregexp for safe regexps. This design is shown in Figure 8. We cache decisions for each regexp used by the application because querying the ReDoS oracles is expensive (0.2 seconds per query, unoptimized).

Our implementation adds 800 lines of C++ to the V8 JavaScript engine and links Node.cure against the RE2 library.

![Decision tree for using RE2 vs. the existing V8 regexp engines.](image)

**Fig. 8.** Decision tree for using RE2 vs. the existing V8 regexp engines. The paths of the regexps from our ReDoS oracle study and npm study corporuses are indicated in the matrices (ReDoS oracle corpus from Synk.io on the left, and npm study corpus on the right). The numerator tracks the regexps known to be truly vulnerable, while the denominator tracks the entire corpus.

**D. Evaluation of our ReDoS defense**

**C-WCET partitioning.** The astute reader will protest that Node.cure’s ReDoS defense is linear-time, not constant-time. First, this is an unavoidable consequence of the JavaScript language specification, which provides synchronous regexps without bound on input length or timeout mechanism. This specification is common to other languages as well. Second, RE2’s linear growth is shallow, so bounding the input string length (even to a large value, e.g. 100KB) at the application or framework level yields a single C-WCET partition. Bounding input length is not an option with Irregexp because exponential behavior can be triggered even with legitimate input lengths (e.g. 30 characters).

**Vulnerabilities, defended and remaining.** As shown in the matrices of Figure 8, 97% (31 of 32) of the potentially-vulnerable regexps from our ReDoS oracle study are protected automatically because they are routed to RE2, including 95% of regexps we know to be vulnerable; and likewise 51% of the potentially-vulnerable regexps from our npm study (2107 of 4122) are protected, including 96% of those we know to be vulnerable (53 of 55). Note the security implications of using ReDoS oracles in this design: if the ReDoS oracles in use have false negatives, vulnerable regexps will be directed to Irregexp. If they have false positives, the “vulnerable” regexps that cannot use RE2 may actually be safe.

Node.cure leaves three avenues for ReDoS-based EHP attacks, corresponding to a weakness in each decision of Figure 8. First, querying the ReDoS oracles is relatively slow. If an attacker can convince the server to evaluate uncached regexps, he can perform a Slow DoS-style EHP attack. Though we don’t anticipate that servers generally evaluate an ever-increasing number of unique regexps, faster ReDoS oracles would defend against this threat. Second, an attacker can exploit any vulnerable regexps that our ReDoS oracles miss (false negatives). This threat can be addressed with more accurate oracles, or by timing out Irregexp evaluations and dropping back to RE2. Lastly, though RE2 handles many vulnerable regexps, an attacker can still carry out standard ReDoS attacks on the subset of vulnerable regexps that contain exponential-time features. This is unavoidable under many language specifications.

**Practicality.** During application startup, Node.cure introduces significant performance overheads due to the cost of querying the oracles. Servers do not typically dynamically generate regexps because of the risk of ReDoS, so we anticipate that our approach of caching oracle decisions will allow them to quickly reach a steady-state where the only overhead is the slowdown from the use of RE2\textsuperscript{47}.

**Adaptability.** Our goal in replacing Node.js’s regexp engine was to offer safety without requiring application-level refactoring. Unfortunately, we found that RE2 and Irregexp are not entirely compatible. We used RE2 on the Irregexp test suite and found one regexp on which they disagree. On regexp `/\(?:(a)\|\(b\)\|c)\)/` with input `abc`\textsuperscript{48}, Irregexp returns a single match `c`, while RE2 returns three separate matches `a`, `b`, and `c`\textsuperscript{49}. This slight incompatibility means that applications would need to be ported to Node.cure.

However, Node.cure can be used by real Node.js applications. Node.cure passes V8’s regexp tests\textsuperscript{50}, runs our small Node.js applications, and can also handle more complex Node.js software like npm’s command-line utility.

**Limitations.** As regexp flags are not the root cause of the exponential-time matching, supporting them is orthogonal to defending against EHP attacks. Though Node.cure honors the case-insensitive flag `i`, it refers regexps with other flags to V8’s default regexp engines.

**VII. Case Study: IODoS in Node.js**

In EDA-based frameworks, I/O-bound activities must be performed carefully to avoid poisoning an Event Handler. In this section we expose vulnerabilities in Node.js’s file I/O implementation, describing two I/O Denial of Service (IODoS)-based EHP attacks. We are not aware of prior research that achieves DoS using I/O. Unlike regexps, which are portable

\textsuperscript{44} See https://github.com/google/re2.

\textsuperscript{45} See http://lh3lh3.users.sourceforge.net/reb.shtml.

\textsuperscript{46} See https://developers.google.com/octane/benchmark.

\textsuperscript{47} This cost could be averted by priming the cache of regexp decisions from a database.

\textsuperscript{48} See v8/test/mjsunit/regexp-loop-capture.js.

\textsuperscript{49} Perl and Python agree with RE2.

\textsuperscript{50} The ReDoS oracles label as safe the only regexp on which Irregexp and RE2 disagree, so it is handled by Irregexp per Figure 8.
across operating systems, I/O requires OS-specific system calls, and this section discusses the Node.js implementation on Linux.

The Node.js framework offers two classes of I/O: network I/O and file I/O. I/O operations can be categorized both by where they are performed (Event Loop or Worker Pool) and how they are implemented. Network I/O in Node.js is done on the Event Loop in a non-blocking fashion, using sockets and epoll\footnote{An exception is the DNS lookup API, which is performed quasi-asynchronously.}. Node.js does file I/O in a blocking fashion, either synchronously on the Event Loop (e.g. `read/writeFileSync`) or quasi-asynchronously (blocking but on the Worker Pool, e.g. `read/writeFile`). The APIs for reading files are described in Table IV.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>readFileSync</code></td>
<td>EL</td>
<td>Yes</td>
<td>No</td>
<td>(a) SF, LF</td>
<td></td>
</tr>
<tr>
<td><code>readSync</code></td>
<td>EL</td>
<td>No</td>
<td>No</td>
<td>(a) SF, LF</td>
<td></td>
</tr>
<tr>
<td><code>readFile</code></td>
<td>WP</td>
<td>Yes</td>
<td>No</td>
<td>(b) SF, LF</td>
<td></td>
</tr>
<tr>
<td><code>readStream</code></td>
<td>WP</td>
<td>Yes</td>
<td>Yes</td>
<td>(c) SF</td>
<td></td>
</tr>
<tr>
<td><code>read</code></td>
<td>WP</td>
<td>No</td>
<td>No</td>
<td>(b) SF, LF</td>
<td></td>
</tr>
</tbody>
</table>


In Node.js, file I/O is done either on the Event Loop (fs.XSync) or on the Worker Pool (fs.X). Though I/O on the Event Loop is supported for convenience in developing command-line applications, it is discouraged for server-side programming because of the risk of long-running I/O.

The quasi-asynchronous I/O done on the Worker Pool is safe as long as a critical assumption holds: that an I/O task will not block its Worker for “too long”, i.e. that the complexity of the I/O is relatively small. But I/O can be viewed as an operation with arbitrary complexity that happens to be fast enough in normal cases. The complexity of the I/O can be increased if an attacker can choose the file, as accommodated by our threat model (§III-A).

To perform IODoS using file I/O, an attacker should look for two kinds of files: Large Files and Slow Files. Large Files are large enough to take a noticeable amount of time to read completely. Files in the many-MB to GB range may suffice, so `/var/log` is a good place to look. Slow Files have variable, potentially-slow I/O times. Since files accessed over a network share are often Slow Files, `/mnt` is a good starting point.

Some of the APIs in Table IV are vulnerable when used to read from Large Files, and all are vulnerable when used to read from Slow Files. An API is vulnerable to Large Files when it submits large read requests. For example, when using `readFile` (an un-partitioned whole-file API) to read a Large File, `readFile` will submit a single read request spanning the entire file, blocking its Event Handler for several seconds when reading even a local Large File. When reading from a Slow File, the partitioned APIs will block the Event Handler during each request, and the whole-file APIs will take far longer\footnote{This is a file system-based analogue to HTTP Slow DoS attacks \cite{8}.}. Using the language of §IV, the whole-file APIs (`readFileSync, readFile`) are not partitioned, and though a fine-grained API may be partitioned in size, it cannot be partitioned in “complexity” (time); smaller reads do not protect the Event Handler from a Slow File’s variability. A sample IODoS-based EHP attack, exploiting unsanitized input to `readFile`, is described in Figure 4.

These Large File and Slow File forms of IODoS are possible in the EDA, but not the OTPCA. If I/O is performed on a thread dedicated to the client it serves, that thread will simply block on I/O, yielding to the threads dedicated to other clients. IODoS attacks become possible in the EDA when Event Handlers are dedicated synchronously to the I/O.

\textbf{B. RA6: Defeating IODoS attacks with partitioning and KAIO}

As file I/O is an integral part of many services, providing safe asynchronous file I/O should be the responsibility of a server-side EDA framework. When performing I/O, developers should be able to trust their framework. As described in §VII-A, developers using Node.js cannot do so.

In this section, we describe our Node.cure prototype, which extends Node.js to address IODoS from Large Files and Slow Files. Where possible, Node.cure handles Large Files using partitioning (§VII-B1) and Slow Files using KAIO (§VII-B2).

1) Large Files: The `readFileSync, readSync, readFile, and read APIs are potentially vulnerable to IODoS using Large Files. We can do nothing for `readFileSyncSync` and `readSync` because they are synchronous and have no timeout mechanism; like regexps, they must block the Event Loop until the requested I/O completes. We can, however, repair `read` and `readFile`. Partitioning a read API protects it against Large Files. Though Node.js performs the I/O for these APIs in one request, there is no semantic need to do so. Thus, we partition these requests into 64KB blocks\footnote{This is the blocksize used by Node.js’s partitioned `readStream` API.} using the existing implementation of `readStream`, buffering the output until the data requested by the user is available. The effect is to move the implementation from pattern Figure 6 (b) to (c). This simple change required about 20 lines in Node.js’s JavaScript-level file system module.

2) Slow Files: All of the read APIs are vulnerable to IODoS using Slow Files. Though the synchronous APIs are indefensible, here we show how to defend the asynchronous APIs from Slow Files using Linux’s support for Kernel Asynchronous I/O (hereafter, KAIO) to offer true asynchronous I/O.

\textbf{Background on KAIO.} KAIO allows an application to submit I/O requests to the kernel, and to poll for an asynchronous completion notification. This means that reads of Slow Files and normal files can proceed concurrently, so slower requests wouldn’t block faster ones. The kernel can deal with Slow Files more effectively than can userspace, so...
using KAIO for slow files won’t just move the problem of Slow Files to the kernel.

The KAIO interface constrains our design. Linux has an io_submit API to submit an AIO request and an io_getevents request to test for completed I/O. Though applications can repeatedly query io_getevents to test for completion, Linux will also signal a file descriptor supplied with the O_DIRECT flag, which bypasses caching on read buffers of I/O requests must be aligned on sector boundaries.

Node.cure Design. Node.cure could defend applications against Slow Files by performing all asynchronous file I/O using KAIO, but like using RE2 by default (§VI-C) this would have unpleasant effects on performance. Specifically, KAIO may increase file I/O throughput but worsen latency. Throughput may improve because it is currently limited by the size $k$ of the Worker Pool; only $k$ concurrent I/O requests are possible, which may not saturate the I/O layer. However, KAIO will increase the file I/O latency because KAIO requires use of the O_DIRECT flag, which bypasses caching on read and write. In consequence, a KAIO-only solution would offer throughput and IODoS-proofing at the cost of higher latency.

We observe, however, that the IODoS-based EHP concern arises only when Slow Files are accessed. Our design is therefore analogous to our ReDoS defense (Figure 8): we favor the existing fast path unless we have evidence that the file is Slow. While our ReDoS defense relied on a static oracle analysis, our IODoS defense identifies vulnerabilities dynamically using stopwatch logic. If a file’s I/O times are outliers in the stopwatch distribution, we label it as Slow.54 Once a file is labeled as Slow, subsequent accesses use the slow-but-secure KAIO path, on which Node.cure performs I/O using KAIO rather than by offloading to the Worker Pool.

Implementing the stopwatch logic and the slow-but-secure KAIO path added about 500 lines to the libuv library that provides I/O to Node.js.

C. Evaluation of our IODoS defenses

C-WCET partitioning. Our Large Files defense introduces partitioning where possible, but the partitioning is only in size, not complexity. Our Slow Files defense identifies “complex” (Slow) files and partitions them in complexity. Each partition on our slow-but-secure KAIO path is C-WCET, consisting of preparing an appropriate request description and offloading it to the kernel via io_submit. The result is an I/O-flavored C-WCET partitioning, with the partitions handled on either the Worker Pool (fast path) or the Event Loop (KAIO path).

Vulnerabilities, defended and remaining. Our threat model assumes the attacker can control the files read by the vulnerable APIs, as is true of the 145 Directory Traversal vulnerabilities reported earlier (§III-D). Our prototype will defend Node.js applications that use these modules from IODoS-based EHP attacks that rely on Large Files and Slow Files. Note that a more powerful attacker can trigger IODoS using Large Files and Slow Files indirectly as well, e.g. by changing an application’s configuration to store its log files in a network share. The attacker is not interested in the contents of the Large or Slow File, but merely that they be used.

One attack we do not address is a special case of Large Files. If an attacker can identify and read from a Bottomless File, in fulfilling the I/O request the server will eventually run out of memory and crash. We view Bottomless Files as a special case with which a framework should not concern itself, since applications can have legitimate reasons for reading from them. As examples of Bottomless Files, we suggest device files like /dev/zero and named FIFOs. Linux has only a few Bottomless device files, and named FIFOs are not usually present on default installations, so an application-level policy-based blacklist seems reasonable [18].

Demonstration of effectiveness. We used a microbenchmark to show the effectiveness of Node.cure’s IODoS defenses. Our benchmark first simulates an IODoS-based EHP attack, poisoning the Worker Pool with $k$ readFile requests of a 2MB Slow File55. After launching the I/O requests, the benchmark executes a CPU-bound workload consisting of 100 zip requests submitted serially. Node.js offloads these zip requests to the Worker Pool, where they compete for Worker processing time with the ongoing slow read requests.

In Figure 9 we compare how quickly the zip requests are fulfilled using Node.js and two forms of Node.cure (with only our Large Files defense, and with both defenses). To eliminate caching effects, we used O_DIRECT for all reads. The EHP attack proves effective against Node.js; because readFile is not partitioned, one of the Workers must complete its entire read before the zip workload’s tasks will have a turn. At around 15 seconds one of the reads completes, after which the zip workload can complete. With Node.cure (Large Files defense), the zip requests interleave with the partitioned slow reads, and the zip workload is able to make consistent progress. This is the same performance that would be seen by an application using vanilla Node.js and the partitioned readStream API. With Node.cure (both defenses), we detect the Slow File after the first readFile request completes, performing the subsequent partitioned reads using KAIO and allowing the zip workload to complete quickly.

54In our prototype, we use a simple threshold of 500 ms.

55Using the same techniques as in §III-C, we gave this file a 100 KB/second I/O rate for illustrative purposes.
because the just-opened inode is cached. The inode number obtained from `fstat` is compared against an in-memory Slow File Inodes hash table. Our stopwatch logic wraps read calls with `gettimeofday`. I/O performed on the KAIO path incurs an additional copy and potentially additional I/O because requests must be aligned, but is less costly than reading from a Slow File directly. Overall, the overhead is small compared to the cost of the I/O being performed.

**Adoptability.** We believe Node.js is adoptable. It adds minimal overhead, retains Node.js semantics, and can protect Node.js applications from IODoS due to Slow or Large Files.

**Limitations.** We draw attention to two limitations of our prototype. First, our prototype only covers the read path, but our design is equally applicable to the write path. Second, in addition to serving as Bottomless Files, device files like `/dev/random` and `/dev/tty` can be used as Slow Files. As Linux KAIO does not support device files, reads from these files will poison the Worker Pool. Just as for Bottomless Files, an application-level policy seems a good solution. Blacklisting `/dev` is straightforward.

**VIII. Discussion**

**In defense of C-WCET partitioning.** We argued earlier (§IV) that C-WCET partitioning will defend EDA applications against EHP attacks. We believe C-WCET partitioning is a practical design principle, and have already shown its applicability for ReDoS (§VI) and IODoS (§VII). At the framework level, the implementation of Node.cure was not difficult once we had C-WCET partitioning as a guide. At the application level, C-WCET partitioning can be implemented using existing techniques to construct asynchronous algorithms (promises, async/await, etc.), though some caution is required to avoid concurrency errors [37], [16]. A tool to check EDA-based applications for C-WCET partitioning would make interesting future work, as would a tool that performs C-WCET partitioning automatically.

In §IV we discussed the difficulties associated with some other defenses against EHP. Discarding functionality costs generality, traditional input sanitization is piecemeal, and timeouts lead to difficult policy decisions and have previously been shown ineffective in preventing DoS attacks [15]. While there are alternative styles of defense, they lose the principle benefit of the EDA. Significantly increasing the size of the Worker Pool, incorporating runtime-enforced preemption, and performing speculative concurrent execution [10] would all defeat EHP attacks, but each is a variation on the same theme: dedicating resources to each client. As discussed throughout this paper, the OTPPCA is generically safe from EHP attacks, but this comes at the cost of scalability. If the server community wishes to use the EDA, which offers high responsiveness and scalability through the use of cooperative multitasking, we believe C-WCET partitioning is the answer.

Disastrous Portability. The pursuit of portability can be myopic. We found that the Node.js framework favors simpler implementations (regexp) and “lowest common denominator” portability ( quasi-asynchronous file I/O) to simplify maintenance and reduce the size of their code bases. Since this choice exposes Node.js applications to EHP attacks, it is a false economy.

We believe that EDA framework developers must acknowledge EHP and provide applications with well-defined WCET where possible. Avoiding DoS attacks is well worth the cost of maintaining platform-specific implementations of the same functionality. While EHP can be avoided by applications that correctly sanitize user input, they do not always do so. The burden falls to framework developers to address security vulnerabilities where they can.

**Generality.** EHP attacks apply to any EDA-based framework. Table I lists many.

ReDoS-based EHP attacks can be carried out when an EDA-based application uses vulnerable regexps, and especially when its regexp engine uses backtracking unnecessarily. We have identified unnecessary exponential-time worst-case behavior in the regexp engines of JavaScript, Perl, Python, Ruby, and Java, and expect similar vulnerabilities in other commonly-used regexp engines. Our hybrid regexp design could be applied to each of these engines, and can be used as a guide for other CPU-bound operations.

IODoS-based EHP vulnerabilities arise when an EDA framework implements an I/O-bound operation synchronously or quasi-asynchronously (offloading it to the Worker Pool). This is how many popular frameworks implement file I/O, including Node.js (JavaScript) and libuv (C/C++). Our KAIO-based defense can be applied to any of these frameworks, and can be used as a guide for other I/O-bound operations.

Kernel support for true asynchronous I/O is available in several operating systems, e.g. Windows’s IOCP/ReadFileEx and Solaris’s aio_read. Other operating systems (e.g. OSX) are indefensible through this means.

**Desirable EDA framework features.** Space limitations induce brevity, but our suggestions for EDA framework design to reduce the risk of EHP include: incorporating asynchronicity even for everyday language features like regexps; adding timeouts as a possible outcome of asynchronous calls; making asynchronous calls cancellable; and using OS-specific support for asynchronicity.

**IX. Related Work**

Throughout the paper we have referred to specific related work. Here we place our work in its broader context, situated at the intersection of the Denial of Service literature and the Event-Driven Architecture community.

**Denial of Service attacks.** Research on DoS can be broadly divided into network-level attacks (e.g. distributed DoS attacks) and application-level attacks [2]. Since EHP attacks exploit the semantics of the application, they are application-level attacks, not easily defeated by network-level defenses.

DoS attacks seek to exhaust the resources critical to the proper operation of a server, and various kinds of exhaustion have been considered. The brunt of the literature has focused on exhausting the CPU. Some have examined the gap between common-case and worst-case performance in algorithms and data structures, for example in quicksort [38], hashing [15], and exponentially backtracking algorithms like ReDoS [14].

---

56For Node.js, see e.g. https://github.com/nodejs/node-v0.x-archive/issues/1446 and https://groups.google.com/forum/#!topic/nodejs/c_R9jIWFbyO.
[47] and rule matching [49]. Other CPU-centric DoS attacks have targeted more general programming issues, showing how to trigger infinite recursion [11] and how to trigger [50] or detect [7] infinite loops. But the CPU is not the only vulnerable resource, as shown by Olivo et al.’s work polluting databases to increase the cost of queries [42]. We are not aware of prior research work that incurs DoS using the file system, as do our Large/Slow File IODoS attacks, though we have found a handful of CVE reports to this effect71.

Our work identifies and shows how to exploit a newly limited resource: Event Handlers. The OTPCA approach ensures that for production-grade web services, attackers must submit enough queries to overwhelm the operating system’s preemptive scheduler, no mean feat. In contrast, in the EDA there are only a few Event Handlers that need be poisoned. Although we prove our point using previously-reported attacks like ReDoS, the underlying resource we are exhausting is the small, fixed-size set of Event Handlers deployed in EDA-based services.

Our work also takes an unusual tack in the application-level DoS space by providing defenses. Because application-level vulnerabilities rely on the semantics of the service being exploited, defenses must often be prepared on a case-by-case basis. In consequence, many works on application-level DoS focus on providing detectors (e.g. [11], [50]), not defenses. When the application-level exploits take advantage of an underlying library, however, researchers have offered replacement libraries (e.g. [15]). Since the EHP attacks we studied in detail (ReDoS and IODoS) are sometimes due to implementation decisions in the Node.js framework, our Node.cure prototype describes a framework-level defense.

**Security in the EDA.** Though researchers have studied security vulnerabilities in different EDA frameworks, most prominently client-side applications, the security properties of the EDA itself have not been researched in detail. Some examples include security studies on client-side JavaScript/Web [33], [30], [17], [39] and Java/Android [22], [21], [6], [29] applications. These have focused on language issues (e.g., eval in JavaScript [51]) or platform-specific issues (e.g., DOM in web browsers [33]). We note that the EDA complicates static analysis because it requires inferring control flows across event callbacks [4], [27], [56], [23].

To the best of our knowledge, there has been little academic engagement with the security aspects of server-side EDA frameworks. Ojamaa et al. [40] performed a Node.js-specific high-level survey, and DeGroef et al. [18] proposed a method to securely integrate third-party modules from npm.

**ReDoS.** The observation that only exponential-time regexp engines have been made before [13], but we are not aware of previous work measuring the size of the opportunity or actually implementing a hybrid regexp engine. Several npm modules offer interfaces to alternative regexp engines, including node-re2 and node-oniguruma. While node-re2 offers a module-

level interface to RE2, node-oniguruma will merely perform exponential-time [34] evaluations on the Worker Pool, still an EHP vulnerability. These modules have not seen broad adoption60, perhaps because unlike our solution they require application refactoring.

**X. Reproduction of Results**

A git repository, available at **BLINDED**, contains everything needed to reproduce our results: analyses of the Snyk.io vulnerability database, the CPU-intensive npm modules, the ReDoS dataset, and the source code for Node.cure.

**XI. Conclusion**

The Event-Driven Architecture holds significant promise for scaling, and it is increasingly popular in the software development community. In this work we defined an EHP attack, a novel class of DoS attack that exploits the cooperative multitasking at the heart of the EDA. We showed that there are EHP attacks hidden among previously-reported vulnerabilities, and observed that the EDA community is incautious about the risks of EHP attacks. We proposed a C-WCET partitioning pattern as an antidote, and found it to be effective in case studies on ReDoS and IODoS. Though our Node.cure prototype addressed two framework-level vulnerabilities in Node.js, we are concerned that with the rise in the popularity of the EDA will come many more EHP vulnerabilities at the framework and application levels. More work is needed on this important problem to guide practitioners as they build the next generation of web services.

**REFERENCES**


[58]See https://github.com/uhop/node-re2.


[60]npm reported reported 500 downloads in July 2017 for node-re2 and 20,000 for node-oniguruma.