Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BackREST

François Gauthier ✉
Oracle Labs, Australia

Behnaz Hassanshahi ✉
Oracle Labs, Australia

Benjamin Selwyn-Smith ✉
Oracle Labs, Australia

Trong Nhan Mai ✉
Oracle Labs, Australia

Max Schlüter ✉
Oracle Labs, Australia

Micah Williams ✉
Oracle, USA

Abstract

Following the advent of the American Fuzzy Lop (AFL), fuzzing had a surge in popularity, and modern day fuzzers range from simple blackbox random input generators to complex whitebox concolic frameworks that are capable of deep program introspection. Web application fuzzers, however, did not benefit from the tremendous advancements in fuzzing for binary programs and remain largely blackbox in nature. In this experience paper, we show how techniques like state-aware crawling, type inference, coverage and taint analysis can be integrated with a black-box fuzzer to find *more* critical vulnerabilities, *faster* (speedups between 7.4× and 25.9×). Comparing BackREST against three other web fuzzers on five large (>500 KLOC) Node.js applications shows how it consistently achieves comparable coverage while reporting more vulnerabilities than state-of-the-art. Finally, using BackREST, we uncovered eight 0-days, out of which six were not reported by any other fuzzer. All the 0-days have been disclosed and most are now public, including two in the highly popular Sequelize and Mongodb libraries.

2012 ACM Subject Classification Security and privacy → Web application security

Keywords and phrases Taint analysis, fuzzing, crawler, Node.js

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.29

Related Version *Previous Version*: `https://arxiv.org/abs/2108.08455`

1 Introduction

Fuzzing encompasses techniques and tools to automatically identify vulnerabilities in programs by sending malformed or malicious inputs and monitoring abnormal behaviours. Nowadays, fuzzers come in three major shades: blackbox, greybox, and whitebox, according to how much visibility they have into the program internals [42]. Greybox fuzzing, which was made popular by the AFL fuzzer, combines blackbox with lightweight whitebox techniques, and have proven to be very effective at fuzzing programs that operate on binary input [107, 26, 25].

Most web application fuzzers that are used in practice are still blackbox [4, 7, 2, 79], and, despite decades of development, still struggle to automatically detect well studied vulnerabilities such as SQLi, and XSS [84]. As a result, security testing teams have to invest significant manual efforts into building models of the application and driving the fuzzer to trigger vulnerabilities. To overcome the limitations of current blackbox web application fuzzers, and find more vulnerabilities automatically, new strategies must be investigated. To pave the way for the next generation of practical web application fuzzers, this paper first shows how REST-like API models are a suitable abstraction for web applications and how adding lightweight coverage and taint feedback loops to a blackbox fuzzer can significantly improve performance and detection capabilities. Then, it highlights how the resulting BackREST greybox fuzzer consistently detects more vulnerabilities than state-of-the-art. Finally, our evaluation reveals how BackREST found eight 0-days, out of which six were *missed* by all the web application fuzzers we compared it against.

API model inference Model-based fuzzers, which use a model to impose *constraints* on input, dominate the web application fuzzing scene. Existing model-based web application fuzzers typically use dynamically captured traffic to derive a base model, which can be further enhanced manually [5, 2, 4]. As with any dynamic analysis, relying on captured traffic makes the quality of the model dependent on the quality of the traffic generator, be it a human being, a test suite, or a crawler. To navigate JavaScript-heavy applications and trigger a maximum number of endpoints, BackREST directs a state-aware crawler towards JavaScript functions that trigger server-side interactions. For completeness, a static type inference analysis is then used to complement the dynamic model.

Feedback-driven What makes greybox fuzzers so efficient is the feedback loop between the lightweight whitebox analysis components, and the blackbox input generator. BackREST is the first web application fuzzer that can focus the fuzzing session on those areas of the application that have not been exercised yet (i.e. by using coverage feedback), and that have a higher chance of containing security vulnerabilities (i.e. by using taint feedback). Taint feedback in BackREST further reports the *type* (e.g. SQLi, XSS, or command injection) of potential vulnerabilities, enabling BackREST to aggressively fuzz a given endpoint with vulnerability-specific payloads, yielding tremendous performance improvements in practice.

Validated in practice The two main metrics that drive practical adoption of a fuzzer are the number of vulnerabilities it can detect and the time required to discover them. Our experiments show how BackREST uncovered eight 0-days in five large ($>$ 500KLOC) Node.js applications, and how adding lightweight whitebox analyses significantly speeds up (7.4-25.9$\times$ faster) the fuzzing session. All 0-days have been disclosed, and most have been announced as NPM advisories, meaning that developers will be alerted about them when they update their systems. Four of them have been tagged with high or critical severity by independent third-parties, and two have been reported against the highly popular `sequelize` (648 745 weekly downloads) and `mongodb` (1 671 653 weekly downloads) libraries.

Contributions This paper makes the following contributions:

- We show how REST-like APIs can effectively model the server-side of modern web applications, and quantify how coverage and taint feedback enhance coverage, performance, and vulnerability detection.
- We empirically evaluate BackREST on five large (>500 KLOC) Node.js (JavaScript) web applications; a platform and language that are notoriously difficult to analyse, and under-represented in current security literature.
- We compare BackREST against three state-of-the-art fuzzers (Arachni, Zap, and w3af)

and open-source our test harness [1].

- We show how greybox fuzzing for web applications allows to detect *severe* 0-days that are missed by all the blackbox fuzzers we evaluated.

Takeaways In our industrial setting, black-box web application fuzzing is used as a security testing tool, where the goal is to automatically detect a maximum number of security bugs and security regressions in a limited amount of time (e.g. a nightly test). With BackREST, we show that extending a black-box web application fuzzer with simple grey-box analyses that use coverage and taint feedback to *skip* and *select* rather than *derive* new inputs can reduce runtime while increasing the number of reported bugs. In our case, the investment in development time (i.e. to extend an existing black-box fuzzer) was quickly dwarfed by the time saved during each fuzzing campaign.

The rest of this paper is structured as follows. Section 2 presents our novel API model inference technique. Section 3 and Section 4 detail the BackREST feedback-driven fuzzing algorithm and implementation, respectively. Section 5 evaluates BackREST in terms of coverage, performance, and detected vulnerabilities and compares it against state-of-the-art web application fuzzers. Section 6 presents and explains reported 0-days. Sections 7 and 8 present related work and conclude the paper.

2 API Model Inference

Web applications expose entry points in the form of URLs that clients can interact with via the HTTP protocol. However, the HTTP protocol specifies only how a client and a server can send and receive information over a network connection, not how to structure the interactions. Nowadays, representational state transfer (REST) is the *de facto* protocol that most modern client-side applications use to communicate with their backend server. While the REST protocol was primarily aimed at governing interactions with web services, we make the fundamental observation that client-server interactions in modern web applications can also be modelled as REST-like APIs. Indeed, at its core, REST uses standard HTTP verbs, URLs, and request parameters to define and encapsulate client-server interactions, which, from our experience, is also what many modern web application frameworks do (e.g. Spring (Java), Ruby on Rails (Ruby), Django (Python), and Express.js (JavaScript)).

Despite the plethora of REST-related tools available, the task of creating an initial REST specification remains, however, largely manual. While tools exist to convert captured traffic into a REST specification, the burden of thoroughly exercising the application or augmenting the specification with missing information is still borne by developers. BackREST alleviates this manual effort by extending a state-aware crawler designed for rich client-side JavaScript applications [50] to dynamically infer REST APIs through crawling. Modern web applications, and single-page ones in particular, implement complex and highly interactive functionalities on the client side. A recent Stack Overflow developer survey [10] shows that web applications are increasingly built using complex client-side frameworks, such as AngularJS [8] and React [9]. In fact, three out of five applications we evaluate in Section 5 heavily use such frameworks. Using a state-aware crawler that can automatically navigate complex client-side frameworks allows BackREST to discover server-side endpoints that can only be triggered through complex JavaScript interactions.

[1] `https://github.com/uqcyber/NodeJSFuzzing`

2.1 Motivating Example

Listing 1 shows an endpoint definition from a Node.js Express application. At line 1, `app` refers to the programmatic web application object, and the `delete` method is used to define an HTTP `DELETE` entry point. The arguments to `delete` include the URL (i.e. `"/users/"`) and path parameter (i.e. `":userId"`) of the entry point, and the callback function that will be executed on incoming requests. The callback function receives request (i.e. `req`) and response (i.e. `res`) objects as arguments, reads the `userId` path parameter at line 2, and removes the corresponding document from a collection in the database at line 3, and leaves the response untouched. The API specification, in OpenAPI format [93], corresponding to the example entry point of Listing 1 is shown in Listing 2. Lines 2-4 define the `"paths"` entry that lists valid URL paths, where each path contains one entry per valid HTTP method. Lines 5-13 define the `"parameters"` object that lists the valid parameters for a given path and HTTP method. Specifically, line 7 defines the name of the parameter, line 8 specifies that it is a path parameter, line 9 specifies that the parameter is required, line 10 specifies that the expected type of the `"userId"` parameter is `"string"` and line 11 captures a concrete example value that was observed while crawling. Directing the crawler to exercise client-side code that will trigger server-side endpoints is, however, non-trivial and covered in the next section.

```
1 app.delete("/users/:userId", (req, res) => {
2   const id = req.params.userId;
3   collection.remove({"id": id});
4 });
```

Listing 1 Example endpoint and its callback in Express

```
 1 {
 2   "paths": {
 3     "/users/{userId}": {
 4       "delete": {
 5         "parameters": [
 6           {
 7             "name": "userId",
 8             "in": "path",
 9             "required": true,
10             "type": "string",
11             "example": "abc123"
12           }
13         ]
14       }
15     }
16   }
17 }
```

Listing 2 Automatically generated OpenAPI specification for the endpoint in Listing 1

2.2 Prioritised State-Aware Crawling for API Inference

To improve responsiveness, modern web applications often transfer a large portion of their logic, including data pre-processing and validation to the client side. To prevent errors, frameworks implement checks on the server-side that validate the structure and to some extent the content of incoming requests. Fuzzing modern web applications thus requires us to produce requests that get past those initial server-side checks. To this end, BackREST infers APIs from the requests generated by a state-aware crawler. Our crawler extends the one presented in [50] to automatically discover endpoints, parameters and types, and to produce concrete examples, as shown in Listing 2.

Figure 1 shows the architecture of the crawler in [50]. It combines prioritized link extraction (or spidering) and state-aware crawling to support both multi-page and single-page applications. It performs static link extraction and dynamic state processing using tasks that are managed in parallel. Given the URL of a running instance, the crawler's task manager first creates a link extraction task for the top level URL, where static links are extracted from HTML elements. Next, new link extraction tasks are created recursively for newly discovered links. At the same time, it creates new states for each extracted link and adds them to a priority queue in a state crawl task. The state queue is prioritized either in a Depth First Search (DFS) or Breadth First Search (BFS) order, depending on the structure of the application. For instance, if an application is a traditional multi-page application and most of the endpoints would be triggered through static links, BFS can be more suitable, whereas a single-page application with workflows involving long sequences of client-side interactions, such as filling input elements and clicking buttons, would benefit more from a DFS traversal.

A state in the crawler includes the URL of the loaded page, its DOM tree, static links and valid events. To avoid revisiting same states, it compares the URLs and their DOM trees using a given distance threshold. As described in [50], it marks a state as previously visited if there exists a state in the cache that has the same URL and a DOM tree that is similar enough to the existing state. To compare the DOM trees, it parses them using the ElementTree XML [11] library and considers two trees to be in the same equivalence class if the number of different nodes does not exceed a threshold.

The crawler transitions between states by automatically triggering events that it extracts from HTML elements. It supports both statically and dynamically registered events, as well as customized event registration in frameworks. Determining the priority of events is one of the differentiating factors between state-aware crawlers. On the one hand, Crawljax [73], a well-known crawler for AJAX-driven applications randomly selects events from a state. On the other hand, Artemis [20], selects events that are more likely to increase code coverage. Feedex [74] instead prioritizes events that trigger user-specified workflows, such as adding an item. jÄk [79] uses dynamic analysis to create a navigation graph with dynamically generated URLs and traces that contain runtime information for events. Its crawler then navigates the graph in an attempt to maximise server-side coverage. Similar to jÄk, we also perform dynamic analysis to detect dynamically registered events that are difficult to detect statically and maximise coverage of server-side endpoints.

Contrary to jÄk, however, our crawler also uses the JavaScript call graph analysis by Feldthaus et al. [35] to compute a distance metric from event handler functions to target functions like `XMLHttpRequest.send()`, `HTMLFormElement.submit()`, or `fetch` and prioritize events that have smaller distances. While the call graph is not sound nor precise, it is being refined as the application is being crawled, allowing the crawler to reach increasingly deeply embedded target functions in the application code and the frameworks and libraries it uses. This prioritization strategy allows BackREST to maximize coverage of JavaScript functions that trigger server-side endpoints. For more details about the refined JavaScript call graph and the prioritization strategy in the crawler see [50]. Listing 3 shows a hande-made and very simplified code-snippet that uses AngularJS for event handling. While the first button, `B1` is a normal button and uses standard `click` event registration, the second button, `B2` uses a custom event registration from the AngularJS framework. Our crawler can successfully correlate the

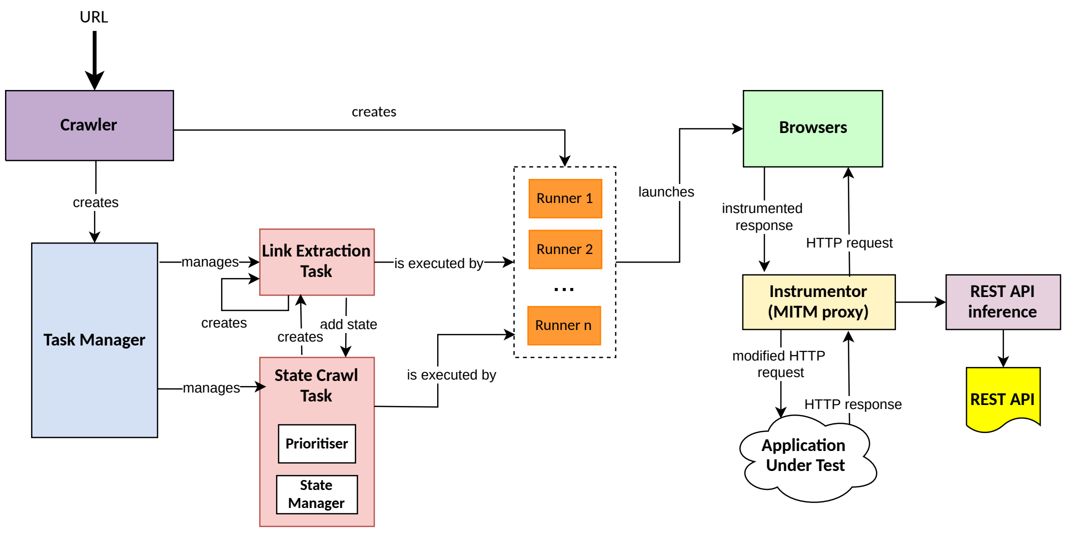


Figure 1 State-aware crawler architecture

customized `click` event for button `B2` and prioritize `B2` over `B1` to call `fetch("users/abc123")` and trigger the server-side `users/abc123` endpoint.

```
<html>
  <script src='angular.js'></script>
  <script>
    function foo(){
    }
    function bar() {
      fetch("users/abc123");
    }
  </script>
  <body>
    <button id='b1' click='foo()'>B1</button>
    <button id='b2' data-ng-click='bar()'>B2</button>
  </body>
</html>

```

Listing 3 Client-side JavaScript code that uses a framework (Angular.js) with customized event handling and registration.

One of the challenges in dynamic analysis of web applications is performing authentication and correctly maintaining the authenticated sessions. Our crawler provides support for a wide variety of authentication mechanisms, including Single Sign On, using a record-and-replay mechanism. We require the user to record the authentication once and use it to authenticate the application as many times as required. To verify whether a session is valid, we ask the user to provide an endpoint and pattern to look up in the response content once and before the crawling starts. For instance, for Juice-shop (see Section 5), we verify the session by sending a GET request to the `rest/user/whoami` endpoint and check if `"admin@juice-sh.op"` is present in its response content periodically to make sure it is logged in.

We intercept the requests triggered by our crawler using a Man-In-The-Middle (MITM) proxy. Next, we process the recorded HTTP requests to infer an API specification automatically. Because we use a client-side crawler to trigger the endpoints, the recorded traffic contains valid headers and parameter values that are persisted in the API and reused in the fuzzing phase. These seed values can often prove invaluable to get past server-side value checks. Our API inference also aggregates concrete values to infer their types. Going back to the example in Listing 1, by automatically triggering `delete` requests to `/users/` endpoint with an actual `userId` string value (e.g. `abc123`), our API inference adds `userId` path parameter with `string` type to the specification based on the observed `userId` values.

2.2.1 Augmenting Crawled APIs with Static Type Inference

By definition, crawled APIs only capture those endpoints and parameters that were exercised dynamically, meaning that they typically under-approximate the real API of an application. We thus *optionally* complement crawled APIs with statically inferred endpoints and parameters. While static analysis can, in theory, over-approximate the real API of an application, precise static analysis of an entire web application stack is practically infeasible without the help of *stubs*, *mocks*, models, or over- and under-approximate clients [14, 61, 70, 101]. Our work is no exception, and we overcome the challenge of statically analysing `Node.js` web applications through the use of mock request and response objects, combined with an approximate use-based type inference analysis.

The inference analysis starts from application endpoints (see Listing 1), where it collects the declared endpoint URL and path parameters. It then initiates a use-based analysis that populates a mock request object with additional parameters that are *read* from the request, without being explicitly declared as path parameters. Once the mock request is populated with parameters, another analysis infers parameter types from their use.

We now illustrate the inner workings of the inference analysis by revisiting the example in Listing 1. The use-based analysis builds on top of a static type inference analysis [45] that approximates, in an unsound way, the runtime structural types of the elements in a program. In Figure 2, items on the left-hand side of the dashed line show the initial type propagation graph. Orange boxes represent types, grey circles represent abstract elements of the program, purple boxes represent function calls or field accesses, and black arrows capture the flow of types and data in the program. To help the reader map the elements of Figure 2 to the code of Listing 1, whenever possible, we annotated function types and field accesses with their code definition (e.g. `app.delete`, `req.params`). Starting from the top, the type propagation graph shows that `app.delete` is of type function and takes two parameters: a path of type string and a callback that is typed as a function that takes `req` and `res` objects as arguments. Then, `params` field of the `req` object is accessed, followed by the `userId` field that yields the `id` object. The rest of the graph is derived from the other statements in Listing 1. To collect request parameters and their types, our analysis propagates a mock request object through the type propagation graph at step 2., extends it with used parameters (i.e. parameters that are read from the request object) at step 3. and finally infer the types of used parameters at step 4. This final step uses and extends the type specifications of the `Tern.js` tool [51, 45]. The inferred parameters and types are then merged into crawled APIs.

3 Feedback-driven Fuzzing

BackREST builds on top of Sulley [16], a blackbox fuzzer with built-in networking support, and extends it with support for API parsing and fuzzing, as well as coverage and taint feedback. Figure 3 shows the high-level architecture of BackREST. First, the application under test is crawled, an API is derived (1), and augmented with statically inferred parameters (2). Then, BackREST is invoked on the generated API file (3). The API is then parsed (4) and

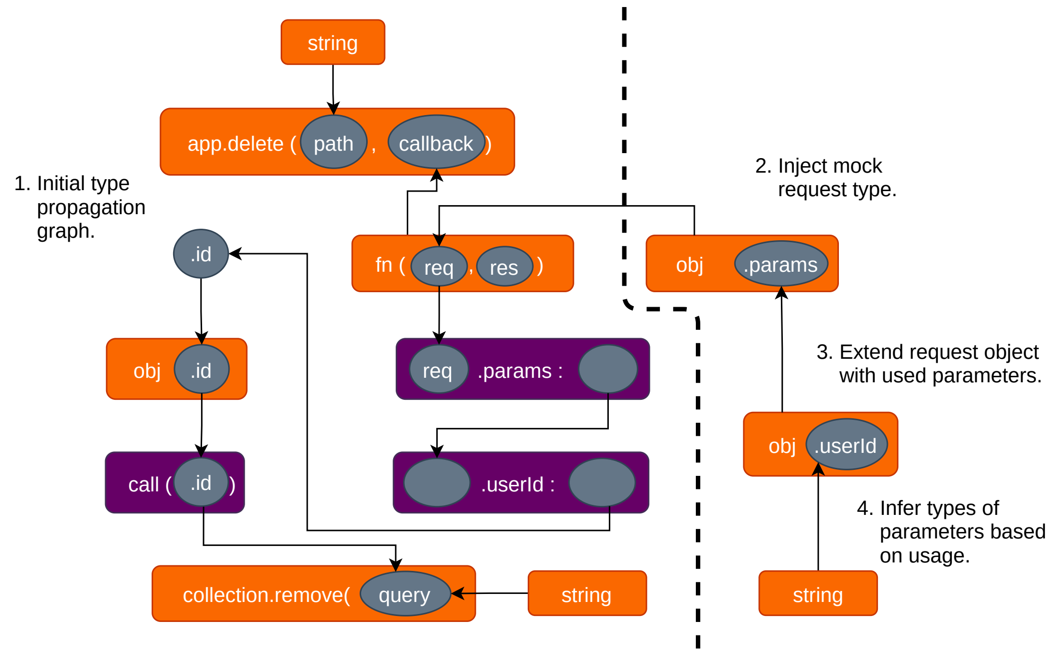

string
app.delete (path , callback)
1. Initial type propagation graph.
.id
fn (req , res)
obj .id
req .params :
call (.id)
.userId :
collection.remove(query
string
2. Inject mock request type.
obj .params
3. Extend request object with used parameters.
obj .userId
4. Infer types of parameters based on usage.
string

Figure 2 Use-based inference of request parameters and types.

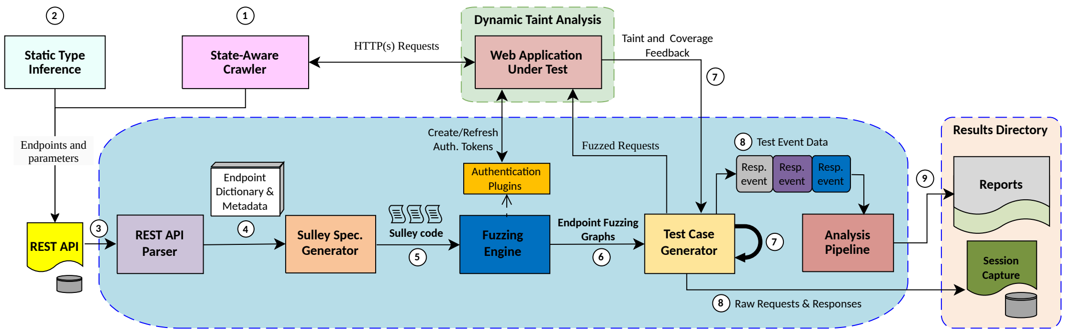


Figure 3 BackREST architecture

broken down into low-level fuzzing code blocks (5). The type information included in the API file is used to select relevant mutation strategies (e.g. string, integer, JWT token, etc.). The fuzzing engine is then responsible for evaluating the individual fuzzing code blocks and reassembling them into a graph that will yield well-formed HTTP requests (6). The test case generator repeatedly traverses the graph to generate concrete HTTP requests, sends them to the application under test, and monitors taint and coverage feedback (7). The HTTP responses are dispatched to the analysis pipeline, which runs in a separate thread, to detect exploits, and are also stored as-is for logging purposes (8). Indicators of exploitation include the response time, the error code and error messages, reflected payloads, and taint feedback. Finally, the analysis results are aggregated and reported (9). To simplify our evaluation setup (section 5), we run BackREST in *deterministic* mode, meaning that it always yields the same sequence of fuzzed HTTP requests for a given configuration.

3.1 Coverage Feedback

Coverage feedback, where the fuzzer uses online coverage information to guide the fuzzing session, was made popular by the AFL fuzzer [107]. Nowadays, most greybox fuzzers use coverage to guide their input generation engine towards producing input that will exercise newly covered code, or branches that will likely lead to new code [66, 65, 36, 80, 64, 26, 25]. Kelinci [57] ported AFL-style greybox fuzzing to Java programs by emulating AFL's coverage analysis and using the AFL fuzzing engine as-is. JQF [77] instead combines QuickCheck-style testing [27] and feedback-directed fuzzing to support inputs with arbitrary data structures. The underlying assumption in AFL and all its derivatives is that targeting code that has not been thoroughly exercised increases the likelihood of triggering bugs and vulnerabilities. Empirical evidence suggests that this assumption holds true for many codebases. Furthermore, the simplicity and widespread availability of coverage analysis makes it suitable for applications written in a wide range of languages.

Compared to mutation-based greybox fuzzers like AFL, BackREST uses coverage information differently. Where AFL-like fuzzers use coverage information to *derive* the next round of input, BackREST uses coverage information to *skip* inputs in the test plan that would likely exercise well-covered code. From that perspective, BackREST uses coverage information as a performance optimisation. Section 3.3 details how BackREST uses coverage information.

3.2 Taint Feedback

Taint-directed fuzzing, where the fuzzer uses taint tracking to locate sections of input that influence values at key program locations (e.g. buffer index, or magic byte checks), was pioneered by Ganesh et al. with the BuzzFuzz fuzzer [37]. In recent years, many more taint-directed greybox fuzzers that build on the ideas of BuzzFuzz have been developed [23, 67, 103, 104, 47, 86].

BackREST uses taint analysis in a different way. With the help of a lightweight dynamic taint inference analysis [41], it reports which input reaches security-sensitive program locations, and the type of vulnerability that could be triggered at each location. Armed with this information, BackREST can prioritise payloads that are more likely to trigger potential vulnerabilities. Taint feedback thus enables BackREST to *zoom in* payloads that are more likely to trigger vulnerabilities, which improves performance and detection capabilities. Taint feedback also improves detection capabilities in cases where exploitation cannot be easily detected in a blackbox manner. Finally, similar to coverage analysis, the relative simplicity of taint inference analysis makes it easy to port to a wide range of languages. The next section details how BackREST uses taint feedback during fuzzing.

3.3 BackREST Fuzzing Algorithm

Algorithm 1 shows the BackREST fuzzing algorithm. It first builds a test plan, based on the API model received as input (line 2). The test plan breaks the API model into a set of *endpoints*, lists "fuzzable" *locations* in each endpoint, and establishes a mutation schedule that specifies the values that are going to be injected at each location. Values are either cloned from the `example` fields, derived using mutations (omitted from Algorithm 1 for readability), or drawn from a pre-defined dictionary of payloads where vulnerability types map to a set of payloads. For example, the SQLi payload set contains strings like: `' OR '1'='1' -`, while the buffer overflow set contains very large strings and numbers.

Algorithm 1 API-based feedback-driven fuzzing

```
Input: web app. $\mathcal{W}$, API model $\mathcal{A}$, threshold $\mathcal{T}$, payload dictionary $\mathcal{D}$
Output: vulnerability report $\mathcal{V}$
$\mathcal{P} \leftarrow \textsc{BuildTestPlan}(\mathcal{A})$
$\mathcal{W}' \leftarrow \textsc{coverageInstrument}(\mathcal{W})$
$\mathcal{W}'' \leftarrow \textsc{taintInstrument}(\mathcal{W}')$
$totalCov \leftarrow 0$
foreach $endpoint$ in $\mathcal{P}$ do
    foreach $location$ in $\mathcal{P}[endpoint]$ do
        $types \leftarrow \mathcal{D}.keys()$
        taint:
        foreach $type$ in $types$ do
            coverage:
            $count \leftarrow 0$
            foreach $payload$ in $\mathcal{D}[type]$ do
                $(resp, currCov, taintCat) \leftarrow \textsc{fuzz}(endpoint, location, payload, \mathcal{W}'')$
                $count \leftarrow count + 1$
                if $currCov > totalCov$ then
                    $count \leftarrow 0$
                end
                $\mathcal{V} \leftarrow \mathcal{V} \cup \textsc{DetectVulnerability}(resp)$
                $totalCov \leftarrow currCov$
                if $taintCat \neq \emptyset$ then
                    $types \leftarrow taintCat$
                    if $type \notin types$ then
                        continue taint
                    end
                    $count \leftarrow 0$
                end
                if $count > \mathcal{T}$ then
                    continue coverage
                end
            end
        end
    end
end
return $\mathcal{V}$
```

```
"/users/{userId}": {
  "delete": {
    "parameters": [
      {
        "name": "userId",
        "in": "path",
        "required": true,
        "type": "string",
        "example": "abc123"
      }
    ]
  }
}
```

Listing 4 Fuzzable locations for an example endpoint

Listing 4 shows an example of an endpoint definition with fuzzable locations in bold. From top to bottom, fuzzable locations include: the value of the `userId` parameter, where the parameter will be injected (e.g. path or request body), whether the parameter is required or optional, and the type of the parameter. Other locations are left untouched to preserve the core structure of the model and increase the likelihood of the request getting past shallow syntactic and semantic checks.

Once the test plan is built, BackREST instruments the application for coverage and taint inference (lines 3-4 of Algorithm 1). Then, it starts iterating over the endpoints in the test plan (line 6). The fuzzer further sends a request for each `(endpoint, location, payload)` combination, collects the response, coverage and taint report, and increases its request counter by one (lines 13-15). If the request covers new code, the request counter is reset to zero, allowing the fuzzer to spend more time fuzzing that particular endpoint, location, and vulnerability type (lines 16-18). The vulnerability detector then inspects the response, searching for indicators of exploitation, and logs potential vulnerabilities (line 19).

Because blackbox vulnerability detectors inspect the response only, they might miss cases where an input reached a security-sensitive sink, without producing an observable side-effect. For example, a command injection vulnerability can be detected in a blackbox fashion only when an input triggers an observable side-effect, such as printing a fuzzer-controlled string, or making the application sleep for a certain amount of time. With taint feedback, however, the fuzzer is informed about: 1. whether parts of the input reached a sink, and 2. the vulnerability type associated with the sink. When the fuzzer is informed that an input reached a sink, it immediately *jumps* to the vulnerability type that matches that of the sink and starts sending payloads of that type only (lines 21-27). The idea behind this heuristic, which we validated on our benchmarks (see Subsection 5.3), is that payloads that match the sink type have a higher chance of triggering observable side-effects to help confirm a potential vulnerability. Targeting specific vulnerability types further minimises the number of inputs required to trigger a vulnerability. Finally, if a given endpoint and location pair have been fuzzed for more than $\mathcal{T}$ requests without increasing coverage, the fuzzer *jumps* to the next vulnerability type (lines 28-30). The idea behind this heuristic, which we also validated on our benchmarks, is that the likelihood of covering new code by fuzzing a given endpoint decreases with the number of requests, unless more complex techniques like symbolic execution are used. In our setup, we set $\mathcal{T}$ to 10 after trying out values in $\{0, 5, 10, 15, 20\}$ and keeping the minimal threshold that detected the maximum number of vulnerabilities. Promptly switching to payloads from a different vulnerability type reduces the total number of requests, thereby improving the overall performance.

Table 1 Benchmark applications

Application	Description	Version	SLOC	Files
Nodegoat	Educational	1.3.0	970 450	12 180
Keystone	CMS	4.0.0	1 393 144	13 891
Apostrophe	CMS	2.0.0	774 203	5 701
Juice-shop	Educational	8.3.0	725 101	7 449
Mongo-express	DB manager	0.51.0	646 403	7 378

4 Implementation

BackREST brings together and builds on top of several existing components. The API inference component uses the state-aware crawler from [50] with an intercepting proxy [28] to generate and capture traffic dynamically. The static API inference uses `Tern.js` [51, 45] to perform type inference. The API parser is derived from PySwagger [6] and the fuzzing infrastructure extends Sulley [16], with API processing support. For coverage instrumentation, we use the Istanbul library [17] and have added a middleware in benchmark applications to read the coverage object after each request, and inject a custom header to communicate coverage results back to the fuzzer. For taint feedback, BackREST implements the Node.js taint analysis from [41] and extends our custom middleware to also communicate taint results back to the fuzzer. The taint analysis is itself built on top of the NodeProf.js instrumentation framework [97] that runs on the GraalVM[2] [3] [106] runtime.

5 Evaluation

In this section, we first review our experimental protocol. Next, we assess the contribution of static type inference to crawled APIs. then, we evaluate how coverage and taint feedback increase the coverage, performance, or number of vulnerabilities detected. The benchmark applications used for evaluation are listed in Table 1. All experiments were run on a machine with 8 Intel Xeon E5-2690 2.60GHz processors and 32GB memory. Then, we compare BackREST to three state-of-the-art web fuzzers. Finally, we present and explain the 0-days that BackREST detected.

5.1 Experimental Design

We took great care to design an empirical evaluation protocol that is fair and adequate. In this section, we review our protocol in the light of the SIGPLAN empirical evaluation guideline [3] and justify divergences from best practices. First, our benchmark applications are all implemented in Node.js and our results might not generalise to web applications implemented in other languages. As a rule of thumb, applications written in languages that have mature instrumentation frameworks will be more easily amenable to the kind of feedback-driven fuzzing implemented in BackREST. Next, whenever we compare BackREST either against itself or other fuzzers, unless otherwise stated (e.g. disabling of certain feedback loops), all runs of BackREST were parameterised in the exact same way. Benchmark-wise, the very

[2] `https://www.graalvm.org/`
[3] GraalVM is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Table 2 Total number of inferred endpoints and parameters. Number of statically inferred values in parenthesis

Benchmark	# Entry points	# Request parameters
Nodegoat	19 (0)	28 (10)
Keystone	20 (0)	69 (46)
Apostrophe	184 (0)	633 (531)
Juice-shop	69 (0)	71 (64)
Mongo-express	29 (0)	96 (49)

Table 3 Impact of the coverage (C) and taint (T) feedback loops on runtime and total coverage, compared to baseline (B)

Benchmark	Coverage (%)			Time (hh:mm:ss)				
	B	C	CT	B	C		CT	
Nodegoat	80.31	78.54	75.59	0:42:39	0:06:07	(7.0×)	00:05:44	(7.4×)
Keystone	48.31	48.05	45.43	5:46:29	0:49:25	(7.0×)	0:13:23	(25.9×)
Apostrophe	—	48.40	45.52	—	11:11:42	–	6:17:34	–
Juice-Shop	74.73	76.34	75.85	12:48:15	1:10:31	(10.9×)	1:08:26	(11.2×)
Mongo-express	69.62	69.57	66.59	2:21:49	0:16:07	(8.8×)	0:11:07	(12.8×)

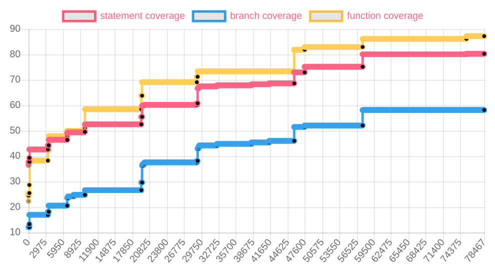
few existing Node.js benchmarks contain libraries only, and are unsuitable for evaluating web application fuzzers. For this reason, we created our own benchmark and open-sourced our evaluation framework to help with reproducibility. Trial-wise, contrary to most fuzzers, we tuned BackREST to be *deterministic*, meaning that every run of the fuzzer produces the same results and that a single trial per experiment is sufficient. Of course, for this to hold, we also assume that applications behave deterministically. To this end, we reset the state of applications after every run, and limit the number of concurrent requests to one. Finally, to ensure a fair comparison of BackREST against other state-of-the-art fuzzers, a co-author of this paper, who did not contribute nor had access to BackREST, was mandated to experiment with and tune the fuzzers to detect a maximum number of vulnerabilities in our benchmark applications.

5.2 API inference

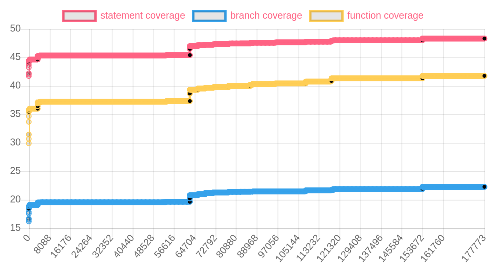
Table 2 shows the number of endpoints and request parameters (excluding path parameters) that were inferred for each benchmark application. The numbers in parenthesis represent the number of additional endpoints and parameters that were identified using static type inference. Results clearly show that our crawler is very efficient at identifying the endpoints of an application while static type inference provides additional request parameters to fuzz.

5.3 Feedback-driven fuzzing

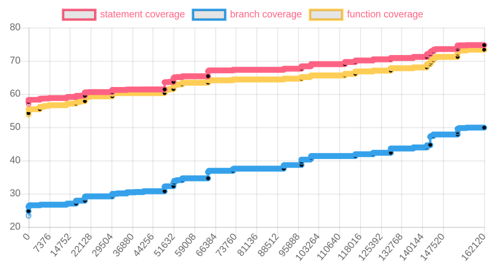
Table 3 compares the total coverage and runtime achieved by enabling the coverage feedback loop only (column C) and combined with taint feedback (column CT) against the baseline blackbox fuzzer (column B). Table 3 also lists speedups for coverage and taint feedback loops compared to baseline. Coverage-wise, enabling the coverage feedback loop, which skips payloads of a given type after $\mathcal{T}$ requests that did not increase coverage, achieves



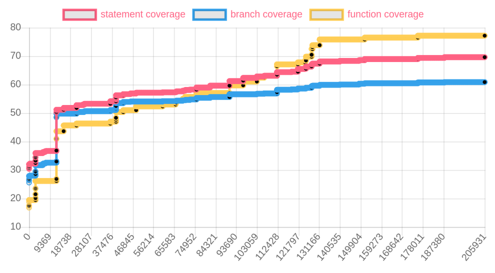
(a) Nodegoat coverage



(b) Keystone coverage



(c) Juice-shop coverage



(d) Mongo-express coverage

Figure 4 Cumulative statement, branch, and function coverage (y axis) in function of the number of requests (x axis) for Nodegoat (a), Keystone (b), Juice-shop (c), and Mongo-express (d) with the baseline blackbox fuzzer

approximately the same coverage (i.e. $\pm$ 2%) in a much faster way (i.e. speedup between 7.0$\times$ and 10.9$\times$). The slight variations in coverage can be explained by many different factors, such as the number of dropped requests, differences in scheduling and number of asynchronous computations, and differences in the application internal state. Indeed, the process of fuzzing puts the application under such a heavy load that exceptional behaviours become more common. Adding taint feedback on top of coverage feedback further decreases runtime, with speedups between 7.4$\times$ and 25.9$\times$. The slightly lower coverage can be explained by the fact that taint feedback forces the fuzzer to skip entire payload types, resulting in lower input diversity and slightly lower total coverage. Finally, the size of the API model for Apostrophe and the load that resulted from using the baseline fuzzer rendered the application unresponsive, and we killed the fuzzing session after 72 hours. Enabling taint feedback for Apostrophe almost halved the runtime compared to coverage feedback alone.

Figure 4 shows the cumulative coverage achieved by the baseline blackbox fuzzer on all applications but Apostrophe. For all applications, cumulative coverage evolves in a step-wise fashion (e.g. marked increases, followed by plateaus) where steps correspond to the fuzzer switching to a new endpoint. The plateaus that follow correspond to the fuzzer looping through its payload dictionary. These results support our coverage feedback heuristic, which is based on the assumption that the likelihood of covering new code by fuzzing a given endpoint decreases with the number of requests.

Table 4 Impact of the coverage (C) and taint (T) feedback loops on bug reports, compared to baseline (B)

Benchmark	(No)SQLi			Cmd injection			XSS			DoS		
	B	C	CT	B	C	CT	B	C	CT	B	C	CT
Nodegoat	0	0	3	0	0	3	5	5	5	0	0	0
Keystone	0	0	0	0	0	0	1	1	0	0	0	0
Apostrophe	0	0	0	0	0	0	1	1	1	2	1	1
Juice-Shop	1	1	2	0	0	1	4	1	1	1	0	0
Mongo-express	0	0	5	0	0	2	0	0	0	3	3	3
Total	1	1	10	0	0	6	11	8	7	6	4	4

5.4 A note on server-side state modelling

Many studies have shown that state-aware crawling of the client-side yields better coverage [73, 20, 74, 79, 32], and our crawler is no exception. Very little is known, however, about the impact of state-aware fuzzing of the server-side. To our knowledge, RESTler [21] is the first study to investigate stateful fuzzing of web services. While the authors have found a positive correlation between stateful fuzzing and increases in coverage, we have not observed a similar effect on our benchmark applications. Similar to RESTler, we attempted to model the state of our benchmark applications by inferring dependencies between endpoints. Specifically, we used the approach from [24] to infer endpoint dependencies from crawling logs and then constrained the fuzzing schedule of BackREST to honour them. This did not improve coverage for all but the Mongo-express application (data not shown). In this particular case, manual inspection revealed that the inferred dependencies were quite intuitive (e.g. insert a document before deleting it) and easily configured.

5.5 Vulnerability detection

Table 4 shows the number of unique true positive bug reports with the baseline fuzzer (column B), with coverage feedback (column C), and further adding taint feedback (column CT). We manually reviewed all reported vulnerabilities and identified the root causes. Table 4 does not list false positives for space and readability reasons. The only false positives were an XSS in Nodegoat that was reported by all three variations, and an SQLi in Keystone that was reported with taint feedback only. Also note that Table 4 lists three types of vulnerabilities (SQLi, command injection, and XSS) and one type of attack (DoS). We opted to list DoS for readability reasons. Indeed, the root causes of DoS are highly diverse (out-of-memory, infinite loops, uncaught exception, etc.) making it difficult to list them all. From our experience, the most prominent root cause for DoS in Node.js are uncaught exceptions. Indeed, contrary to many web servers, the Node.js front-end that listens to incoming requests is single-threaded. Crashing the front-end thread with an uncaught exception thus crashes the entire Node.js process [76].

Interestingly, enabling coverage feedback has no impact on the detection of SQLi and command injection vulnerabilities, suggesting that this optimisation could be enabled at no cost. Enhancing the fuzzer with taint feedback, however, consistently detects as many or more SQLi and command injection vulnerabilities. This is explained by the fact that taint inference does not rely on client-observable side-effects of a payload to detect vulnerabilities. This is especially obvious for command injection vulnerabilities, which are detected with taint feedback only, for which observable side-effects are often hard to correlate to the root cause (e.g. slowdowns, internal server errors), compared to cross-site scripting, for example. Because the most critical injection flaws sit on the server-side of web applications, and are, by nature, harder to detect at the client side, the taint inference in BackREST gives it a tremendous edge over blackbox fuzzers. Table 4 also shows, however, that some cross-site scripting and denial-of-service vulnerabilities are missed when coverage and taint feedback are enabled. First, all missed XSS are stored XSS. Indeed, through sheer brute force, the baseline fuzzer manages to send very specific payloads that exploit stored XSS vulnerabilities and trigger side-effects that can be observed at the client (e.g. reflecting the payload in another page), while coverage and taint feedback loops caused these specific payloads to be skipped. To reliably detect stored XSS, taint analysis would need to track taint flows through storage, which implies shadowing every storage device (i.e. databases and file system) to store and propagate taint. This feature is beyond greybox fuzzing and is known to be tricky to implement, and costly from both time and memory perspectives [58]. Second, the missed denial-of-service vulnerabilities are due to: 1. a slow memory leak that requires several thousand requests to manifest in Apostrophe, and 2. a specific SQLite input that happened to be skipped with coverage and taint feedback.

Evaluating false negatives In the context of a fuzzing session, false negatives are those vulnerabilities that are in the scope of a fuzzer, but that are missed. Accounting for false negatives requires an application with known vulnerabilities. The OWASP Nodegoat and Juice-Shop projects are deliberately vulnerable applications with seeded vulnerabilities. Both projects were built for educational purposes, and both have official solutions available, making it possible to evaluate false negatives. The solutions, however, list vulnerabilities from a penetration tester perspective; they list attack payloads, together with the error messages or screens they should lead to. For this reason, correlating the official solutions to BackREST reports is not trivial. For example, the Juice-Shop solution reports several possible different SQL and NoSQL injection attacks. From a fuzzing perspective, however, all these attacks share the same two root causes: calling specific MongoDB and Sequelize query methods with unsanitised inputs. In other words, while the official solutions report different exploits, BackREST groups them all under the same two vulnerability reports. For this study, we manually correlated all the SQLi, command injection, XSS, and DoS exploits in official solutions to vulnerabilities in the applications and found that BackREST reports them all, achieving a recall of 100% for Nodegoat and Juice-Shop.

5.6 Comparison with state-of-the-art

In this section, we compare BackREST against the arachni [1], w3af [7] and OWASP Zap [4] blackbox web application fuzzers. While we initially planned to also evaluate jÄk [79], our attempts at running it on our benchmark applications ultimately failed because of outdated dependencies (the code is 6 years old), authentication issues, and internal errors. To minimise bias and ensure a fair evaluation, all three remaining fuzzers were evaluated by a co-author of this paper who did not contribute and did not have access to BackREST, and was mandated to tune them to report a maximum number of vulnerabilities. All fuzzers were configured to scan for (No)SQLi, command injection, XSS, and DoS vulnerabilities. Significant care was also taken to configure all the fuzzers to authenticate into the applications and not log themselves out during a scan. Finally, after we discovered that the crawlers in arachni and w3af are fairly limited when it comes to navigating single-page web applications that heavily rely on client-side JavaScript, we evaluated these fuzzers with seed URLs from a

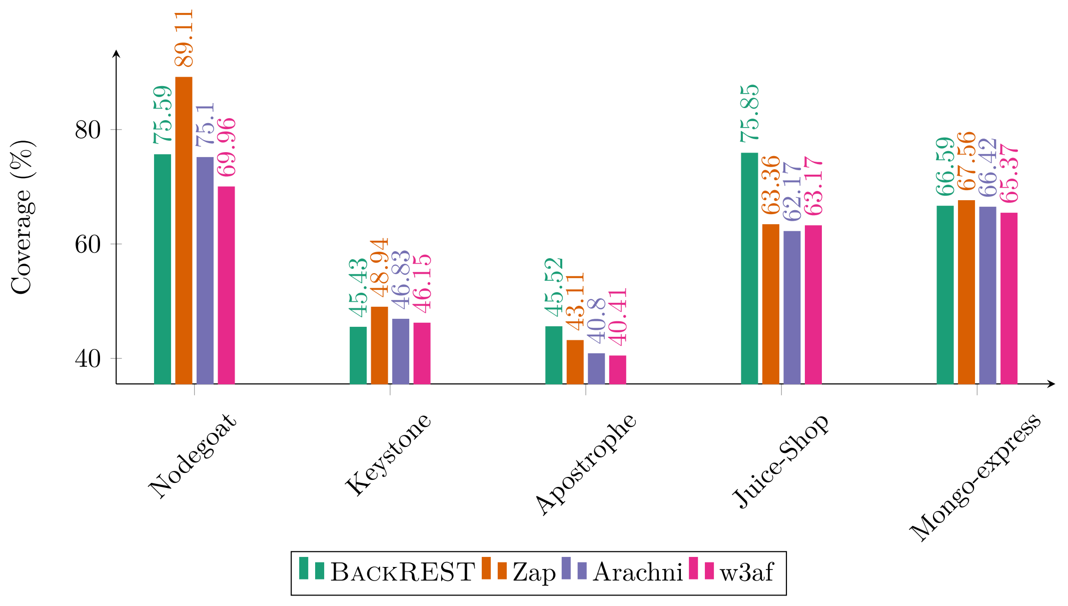


Figure 5 Coverage achieved by different fuzzers.

Zap crawling session. Zap internally uses the Crawljax [73] crawler that is better suited to navigate modern JavaScript-heavy applications.

Figure 5 shows the coverage that was achieved by the different fuzzers on the benchmark applications. BackREST consistently achieves comparable coverage to other fuzzers. Table 6 compares the number of bugs found by each fuzzer. For BackREST, we report the bugs found with coverage and taint feedback only. Apart from denial-of-service, BackREST consistently detects more vulnerabilities than other fuzzers, which we mostly attribute to the taint feedback loop. Indeed, while blackbox fuzzers can only observe the *side-effects* of their attacks through error codes and client-side inspection, BackREST can determine with high precision if a payload reached a sensitive sink and report vulnerabilities that would otherwise be difficult to detect in a purely blackbox fashion. Furthermore, we confirmed through manual inspection that apart from DoS, BackREST always reports a strict superset of the vulnerabilities reported by the other fuzzers. Deeper inspection revealed that the additional DoS found by Zap in Mongo-Express was due to a missing URL-encoded null byte payload (`%00`) in our payload dictionary.

It is very difficult to compare the performance of different web fuzzers, given the number of tunable parameters that each of them offers. For this reason, we focused our efforts on configuring them to maximise their detection power and did let them run until completion. The runtime of BackREST, as reported in Table 3, is directly proportional to the size of the API, which explains the longer runtime on Apostrophe. OWASP ZAP, our closest contender in terms of detected vulnerabilities, took between three minutes and three hours to complete a scan. Arachni took between ten minutes and one hour and a half, and w3af took between one and thirty minutes. Note that the low runtime of w3af is due to the fact that it stops its scan early if the application starts sending too many error responses.

Table 5 0-day vulnerabilities found by BackREST (B), Zap (Z), Arachni (A) and w3af (w3)

Codebase	Vulnerability	Found by	Taint only	Severity	Ref.
MarsDB	Command injection	B	✓	Critical	[39]
Sequelize	Denial-of-Service	B		Moderate	[38]
Apostrophe	Denial-of-Service	B		—	[89]
Apostrophe	Denial-of-Service	B		Low	[40]
Mongo-express[1)]	Command injection	B	✓	Critical	[62]
Mongo-express	Denial-of-Service	B, Z, A, w3		Medium	[71]
Mongodb-query-parser	Command injection	B	✓	Critical	[92]
MongoDB	Denial-of-service	B, Z		High	[49]

1) BackREST independently and concurrently found the same vulnerability

5.7 Reported 0-days

Table 5 lists the 0-days that were identified in benchmark applications and the fuzzers that reported them. The *taint only* column shows whether a particular 0-day was reported with taint feedback *only*. Out of all the vulnerabilities reported in Table 6, nine translated into 0-days, out of which six were reported by BackREST only. Several reasons explain why not all vulnerabilities translated to 0-day. First, recall that Nodegoat and Juice-Shop are deliberately insecure applications with seeded vulnerabilities. While BackREST detected several of them, they are not 0-days. Interestingly, however, BackREST did report non-seeded vulnerabilities in MarsDB, and Sequelize, which happen to be dependencies of Juice-Shop. Through the fuzzing of Juice-Shop, BackREST indeed triggered a command injection vulnerability in MarsDB and a denial-of-service in Sequelize. Second, the XSS that were reported in Apostrophe and Keystone are exploitable only in cases where the JSON response containing the XSS payload is processed and rendered in HTML. While we argue that returning JSON objects containing XSS payloads is a dangerous practice because it puts the consumers of the returned JSON object at risk, developers decided otherwise and did not accept our reports as vulnerabilities. Third, Mongo-express is a database management console; it deliberately lets its users inject arbitrary content. Hence, NoSQLi in Mongo-express can be considered as a *feature*. Otherwise, the command injection and denial-of-service vulnerabilities in Mongo-express and its dependencies all translated into 0-days, and so did the denials-of-services in Apostrophe.

For readers who might not be familiar with the Node.js ecosystem, it is important to underline how the MongoDB and Sequelize libraries are core to *millions* of Node.js applications. At the time of writing, MongoDB[4] had 1 671 653 *weekly* downloads while Sequelize[5] had 648 745. By any standard, these libraries are extremely heavily used, and well exercised.

6 Case studies

In this section, we detail some of the 0-days we reported in Table 5. We also explain some JavaScript constructs that might be puzzling to readers who are not familiar with the

[4] https://www.npmjs.com/package/mongodb
[5] https://www.npmjs.com/package/sequelize

Table 6 Number of vulnerabilities found by BackREST (B), Zap (Z), Arachni (A) and w3af (w3)

Benchmark	(No)SQLi				Cmd inj.				XSS				DoS			
	B	Z	A	w3	B	Z	A	w3	B	Z	A	w3	B	Z	A	w3
Nodegoat	3	3	0	2	3	0	0	3	5	4	2	3	0	0	0	0
Keystone	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Apostrophe	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
Juice-Shop	2	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0
Mongo-express	5	0	0	0	2	0	0	0	0	0	0	0	3	4	3	3
Total	**10**	4	1	3	**6**	0	0	3	**7**	5	2	3	**4**	**4**	3	3

```
1  //Juice-Shop code
2  //Implements the /rest/track-order/{id} route
3  db.orders.find({ $where: "this.orderId === '" + req.params.id + "'" }).then(
4    order => { ... },
5    err => { ... }
6  );
7
8  //MarsDB code
9  $where: function(selectorValue, matcher) {
10   matcher._recordPathUsed('');
11   matcher._hasWhere = true;
12   if (!(selectorValue instanceof Function)) {
13     selectorValue = Function('obj', 'return ' + selectorValue);
14   }
15   return function(doc) {
16     return {result: selectorValue.call(doc, doc)};
17   }
18 };
```

Listing 5 Command injection vulnerability in `MarsDB`

language. All the information presented in the following case studies is publicly available and a fix has been released for all but one of the vulnerabilities we present (MarsDB). In this particular case, the vulnerability report has been public since Nov 5th, 2019.

6.1 MarsDB command injection

MarsDB is an in-memory database that implements the MongoDB API. Listing 5 shows the command injection vulnerability in the MarsDB library that BackREST uncovered. Attacker-controlled input is injected in the client application at line 3, through a request parameter (bolded). The client application then uses the unsanitised tainted input to build a MarsDB `find` query. In Node.js, long-running operations, such as querying a database, are executed asynchronously. In this example, calling the `find` method returns a JavaScript *promise* that will be resolved asynchronously. Calling the `then` method of a promise allows to register handler functions for cases where the promise is fulfilled (line 4) or rejected (line 6). The query eventually reaches the `where` function of the MarsDB library at line 9 as the `selectorValue` argument. That argument is then used at line 13 to dynamically create a new function from a string. From a security perspective, calling the `Function` constructor in JavaScript is roughly equivalent to calling the infamous `eval`; it dynamically creates a function from code supplied as a string. The newly created function is then called at line 16,

```
1  run(sql, parameters) {
2    this.sql = sql;
3    const query = this;
4    function afterExecute(err, results) {
5      ...
6      if (query.sql.indexOf('sqlite_master') !== -1) {
7        if (query.sql.indexOf('SELECT sql FROM sqlite_master WHERE tbl_name') !== -1) {
8          result = results;
9          if (result && result[0] && result[0].sql.indexOf('CONSTRAINT') !== -1) {
10           result = query.parseConstraintsFromSql(results[0].sql);
11         }
12       }
13       else if (results !== undefined) {
14         // Throws a TypeError if results is not an array.
15         result = results.map(resultSet => resultSet.name);
16       }
17       else {
18         result = {}
19       }
20     }
21   }
22 }
```

Listing 6 Denial-of-Service (DoS) vulnerability in `Sequelize`

which triggers the command injection vulnerability [39]. In this particular case, unless the payload is specifically crafted to: 1. generate a string that is valid JavaScript code, and 2. induce a side-effect that is observable from the client, it can be very difficult to detect this vulnerability in a purely blackbox manner. Thanks to taint feedback, BackREST can detect the command injection as soon as *any* unsanitised input reaches the `Function` constructor at line 16.

6.2 Sequelize DoS

Sequelize is a Node.js Object-Relational Mapper (ORM) for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. The code in Listing 6 is vulnerable to a DoS attack that crashes the Node.js server with an uncaught exception. First, an attacker-controllable SQL query is passed as the `sql` argument to the `run` function that executes SQL queries at line 1. The tainted query is assigned to various variables, the query is executed, and after its execution is eventually searched for the string "`sqlite_master`" at line 7, which is a special table in SQLite. If the search is successful, the query is then searched for the string "`SELECT sql FROM sqlite_master WHERE tbl_name`". If this search is unsuccessful and the query returned a `results` object that is not `undefined` (line 14), the `map` [6] method is called on the `results` object at line 16. This is where the DoS vulnerability lies. If the `results` object is not an array, it likely won't have a `map` method in its prototype chain, which will throw an uncaught `TypeError` and crash the Node.js process [38]. In summary, any request that includes the string "`sqlite_master`", but not "`SELECT sql FROM sqlite_master WHERE tbl_name`", and that returns a single value (i.e. not an array), will crash the underlying Node.js process.

[6] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map`

```
self.routes.list = function(req, res) {
  if (req.body.format === 'managePage') {
    ...
  } else if (req.body.format === 'allIds') {
    ...
  }
  return self.listResponse(req, res, err, results);
};
```

Listing 7 Denial-of-Service (DoS) vulnerability in `Apostrophe`

6.3 Apostrophe DoS

Apostrophe is an enterprise content management system (CMS). Listing 7 shows a snippet of Apostrophe code that is vulnerable to a DoS attack. This code reads the `format` parameter of the request body, and checks if it is equal to "`managePage`" or "`allIds`", but misses a fallback option for cases where it is equal to neither. If this situation occurs, an uncaught exception is thrown, crashing the server [89].

6.4 Mongo-express command injections

Mongo-express is a MongoDB database management console. In versions prior to 0.54.0, it was calling an eval-like method with attacker-controllable input, leading to a command injection vulnerability. While BackREST independently detected this vulnerability, it was concurrently reported days before our own disclosure [62]. Interestingly, BackREST then revealed how the fix still enabled command injection. Indeed, the fix was to use the `mongo-db-query-parser` library to parse attacker-controlled input. The issue is that the library is using `eval` itself. Thanks to taint feedback, BackREST detected that tainted input was *still* flowing to an `eval` call, which we disclosed [92].

6.5 MongoDB DoS

```
function createCollection(db, name, options, callback) {
  ...
  executeCommand(db, cmd, finalOptions,
    err => {
      if (err) return handleCallback(callback, err);
      handleCallback(
        callback,
        null,
        // Throws an uncaught MongoError if the name argument is invalid
        new Collection(db, db.s.topology, db.s.databaseName, name,
            db.s.pkFactory, options)
      );
    }
  );
  ...
}
```

Listing 8 Denial-of-Service (DoS) vulnerability in `MongoDB`

MongoDB is a document-based NoSQL database with drivers in several languages. Listing 8 shows a snippet from the MongoDB driver that has a DoS vulnerability. This code gets executed when new collections are created in a MongoDB database. If the name of the collection to be created is attacker-controllable, and the attacker supplies an invalid collection name, the call to the `Collection` constructor at line 10 fails and throws an uncaught `MongoError` that crashes the Node.js process [49]. The taint feedback loop quickly reports that a tainted collection name flows to the `Collection` constructor, enabling BackREST to trigger the vulnerability faster.

7 Related Work

Web application modelling Modelling for web applications has a long and rich history in the software engineering and testing communities. Modelling methods broadly fall into three main categories: graph-based, UML-based, and FSM-based. Graph-based approaches focus on extracting navigational graphs from web applications and applying graph-based algorithms (e.g. strongly connected components, dominators) to gain a better understanding of the application [102, 31]. UML-based approaches further capture the interactions and flows of data between the different components of a web application (e.g. web pages, databases, and server) as a UML model [87, 102, 19]. To facilitate automated test case generation, FSM-based approaches instead cluster an application into sub-systems, model each with a finite-state machine and unify them into a hierarchy of FSMs [18]. All these approaches were designed to model *stateful* web applications, which were the norm back in the early 2000s. Since then, web development practices evolved, developers realised that building server-side applications that are as *stateless* as possible improves maintainability, and the REST protocol, which encourages statelessness, gained significant popularity.

Grammar inference Automated learning of grammars from inputs is a complementary and very promising research area [44, 54, 22]. To efficiently learn a grammar from inputs, however, current approaches either require: 1. very large datasets of input to learn from ([44]); 2. highly-structured parser code that reflects the structure of the underlying grammar ([54]); or 3. a reliable oracle to determine whether a given input is well-formed ([22]). Unfortunately, very few web applications meet any of these criteria, making model inference the only viable alternative. Compared to synthesised grammars, REST models are also easier to interpret.

Model-based fuzzing Model-based fuzzing derives input from a user-supplied [82, 21], or inferred [90, 91, 48, 33, 32] model. Contrary to grammar-based fuzzing [52, 53, 43], input generation in model-based fuzzing is not constrained by a context-free grammar. While grammar-based fuzzers generally excel at fuzzing language parsers, model-based fuzzers are often better suited for higher-level programs with more weakly-structured input.

REST-based fuzzing Most closely related to our work are REST-based fuzzers. In recent years, many HTTP fuzzers have been extended to support REST specifications [59, 85, 98, 83, 100], but received comparatively little attention from the academic community. RESTler [21] is the exception. It uses a user-supplied REST specification as a model for fuzzing REST-based services in a blackbox manner. To better handle stateful services, RESTler enriches its model with inferred dependencies between endpoints. As we highlighted in Subsection 5.4, we haven't found endpoint dependency inference to have a significant impact on the coverage of all but one of our benchmark applications. For Mongo-express, we found that the inferred dependencies were trivial, easily configurable and did not justify the added complexity.

Taint-based fuzzing Taint-based fuzzing uses dynamic taint analysis to monitor the flow of attacker-controlled values in the program under test and to guide the fuzzer [37, 23, 67]. Wang et al. [103, 104] use taint analysis to identify the parts of an input that are compared against checksums. Similarly, Haller et al. use taint analysis to identify inputs that can trigger buffer overflow vulnerabilities [47], while Vuzzer [86] uses it to identify parts of an input that are compared against magic bytes or that trigger error code. Taint analysis has also been used to map input to the parser code that processes it and to infer an input grammar [53]. More closely related to our work, the KameleonFuzz tool [33] uses taint inference on the client-side of web applications to detect reflected inputs and exploit cross-site scripting vulnerabilities. BlackWidow [34] implements several stateful crawling strategies in combination with client-side taint inference to detect stored and reflected cross-site scripting vulnerabilities in multi-page PHP applications. To our knowledge, BackREST is the first web application fuzzer to implement a *server-side* taint inference feedback loop.

Web application security scanning The process of exercising an application with automatically generated malformed, or malicious input, which is nowadays known as fuzzing, is also known as security scanning, attack generation, or vulnerability testing in the web community. Whitebox security scanning tools include the QED tool [72] that uses goal-directed model checking to generate SQLi and XSS attacks for Java web applications. The seminal Ardilla paper [58] presented a whitebox technique to generate SQLi and XSS attacks using taint analysis, symbolic databases, and an attack pattern library. Ardilla was implemented using a modified PHP interpreter, tying it to a specific version of the PHP runtime. Similarly, Wassermann et al. also modified a PHP interpreter to implement a concolic security testing approach [105]. BackREST instrumentation-based analyses decouples it from the runtime, making it easier to maintain over time. More recent work on PHP application security scanning includes Chainsaw [12] and NAVEX [13] that use static analysis to identify vulnerable paths to sinks and concolic testing to generate concrete exploits. Unfortunately, the highly dynamic nature of the JavaScript language makes any kind of static or symbolic analysis extremely difficult. State-of-the-art static analysis approaches can now handle some libraries and small applications [95, 75, 60] but concolic testing engines still struggle to handle more than a thousand lines of code [30, 69, 15]. For this reason, blackbox scanners like OWASP Zap [4], Arachni [1] or w3af [7], which consist of a crawler coupled with a fuzzing component, were the only viable option for security scanning of Node.js web applications. With BackREST, we showed that lightweight coverage and taint inference analyses are well-suited to dynamic languages for which static analysis is still extremely challenging.

Web vulnerability detection and prevention In the past two decades, a very large body of work has focused on detecting and preventing vulnerabilities in web applications. The seminal paper by Huang et al. introduced the WebSSARI tool [55] that used static analysis to detect vulnerabilities and runtime protection to secure potentially vulnerable areas of a PHP application. In their 2005 paper, Livshits and Lam showed how static taint analysis could be used to detect injection vulnerabilities, such as SQLi and XSS, in Java web applications [68]. The Pixy tool [56] then showed how static taint analysis could be ported, to the PHP language to detect web vulnerabilities in PHP web applications. The AMNESIA tool [46] introduced the idea of modelling SQL queries with static analysis and checking them against the model at runtime. This idea was further formalised by Su et al. [96], applied to XSS detection [99] and is still used nowadays to counter injection attacks in Node.js applications [94].

JavaScript vulnerability detection As Web 2.0 technologies gained in popularity, the client-side of web applications became richer, and researchers started investigating the JavaScript code that runs in our browsers. It became quickly obvious, however, that existing static analysis techniques could not be easily ported to JavaScript, and that dynamic techniques were better suited for highly dynamic JavaScript code [88]. Dynamic taint analysis thus started to gain popularity, and was particularly successful at detecting client-side DOM-based XSS vulnerabilities [63, 78]. In the meantime, in 2009, the first release of Node.js, which brings JavaScript to the server-side, came out, and is now powering millions of web applications worldwide. Despite its popularity, however, the Node.js platform comparatively received little attention from the security community [76, 81, 29], with only two studies addressing injection vulnerabilities [94, 75].

8 Conclusion

We presented BackREST, the first fully automated model-based, coverage- and taint-driven greybox fuzzer for web applications. BackREST guides a state-aware crawler to automatically infer REST-like APIs, and uses coverage feedback to avoid fuzzing thoroughly covered code. BackREST makes a novel use of taint feedback to focus the fuzzing session on likely vulnerable areas, guide payload generation, and detect more vulnerabilities. Compared to a baseline version without taint and coverage feedback, BackREST achieved speedups ranging from 7.4× to 25.9×. BackREST also consistently detected more (No)SQLi, command injection, and XSS vulnerabilities than three state-of-the-art web fuzzers and detected six 0-days that were missed by all other fuzzers.

Depending on the context in which fuzzing is used, aspects like runtime, or number, depth, or severity of bugs reported will be prioritised. In our industrial setting, where fuzzing is used as a nightly security testing tool, time is of essence. By extending a blackbox web application fuzzer with coverage and taint feedback loops that helps it *skip* and *select* inputs, we showed how it can detect *more* vulnerabilities *faster*. In our setting, the initial investment in development time was quickly absorbed by the time saved during each fuzzing session, without accounting for the additional bugs found. The analyses we described are simple enough to be applied to a vast number of existing black-box web application fuzzers and we hope that our study will trigger further research in this area.

References

1 Arachni. URL: https://www.arachni-scanner.com/.
2 Burp suite. URL: https://portswigger.net/burp.
3 Empirical Evaluation Guidelines. URL: https://www.sigplan.org/Resources/EmpiricalEvaluation/.
4 OWASP Zed Attack Proxy. URL: https://www.zaproxy.org/.
5 Peach fuzzer community edition. URL: https://www.peach.tech/resources/peachcommunity/.
6 A python client for swagger enabled rest api. URL: https://github.com/pyopenapi/pyswagger.
7 w3af. URL: http://w3af.org/.
8 AngularJS. https://angularjs.org/, 2021. Accessed: 2021-02-1.
9 React.js. https://reactjs.org/, 2021. Accessed: 2021-02-1.
10 Stack Overflow Developer Survey. https://insights.stackoverflow.com/survey/2020#technology-web-frameworks, 2021. Accessed: 2021-02-1.

11 The ElementTree XML library. https://docs.python.org/3/library/xml.etree.elementtree.html, 2021. Accessed: 2021-03-24.

12 Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652, 2016.

13 Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

14 Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 13–18. ACM, 2015.

15 Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Constraint programming for dynamic symbolic execution of javascript. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 1–19. Springer, 2019.

16 Pedram Amini, Aaron Portnoy, and Ryan Sears. Sulley. https://github.com/OpenRCE/sulley.

17 Krishnan Anantheswaran, Corey Farrell, and contributors. Istanbul: Javascript test coverage made simple. URL: https://istanbul.js.org/.

18 Anneliese A Andrews, Jeff Offutt, and Roger T Alexander. Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4(3):326–345, 2005.

19 Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara. Understanding web applications through dynamic analysis. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 120–129. IEEE, 2004.

20 Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of JavaScript Web Applications. In *ICSE*, 2011.

21 Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

22 Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.

23 Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825. IEEE, 2012.

24 Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277, 2011.

25 Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

26 Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

27 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

28 Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 5.1]. URL: https://mitmproxy.org/.

29 James Davis, Arun Thekumparampil, and Dongyoon Lee. Node. fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 145–160, 2017.

30 Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. Type-aware concolic testing of javascript programs. In *Proceedings of the 38th International Conference on Software Engineering*, pages 168–179, 2016.

31 Eugenio Di Sciascio, Francesco M Donini, Marina Mongiello, and Giacomo Piscitelli. Anweb: a system for automatic support to web application verification. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 609–616, 2002.

32 Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 523–538, 2012.

33 Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.

34 Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142, 2021.

35 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

36 Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

37 Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

38 Françcois Gauthier. Denial of Service — sequelize. `https://www.npmjs.com/advisories/1142`.

39 François Gauthier. Command Injection — marsdb. `https://www.npmjs.com/advisories/1122`.

40 François Gauthier. Denial of Service — apostrophe. `https://www.npmjs.com/advisories/1183`.

41 François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. Affogato: Runtime Detection of Injection Attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 94–99. ACM, 2018.

42 Patrice Godefroid. Fuzzing: hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

43 Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

44 Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

45 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices*, 47(6):239–250, 2012.

46 William GJ Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, 2005.

47 Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 49–64, 2013.

48 HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

49 Behnaz Hassanshahi. Denial of Service — mongodb. `https://www.npmjs.com/advisories/1203`.

50 Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *29th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

51 Marijn Haverbeke. A JavaScript code analyzer for deep, cross-editor language support. `https://ternjs.net/`. Accessed: 17-06-2019.

52 Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012.

53 Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725. IEEE, 2016.

54 Matthias Höschele and Andreas Zeller. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 31–34. IEEE Press, 2017.

55 Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

56 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

57 Rody Kersten, Kasper Søe Luckow, and Corina S. Pasareanu. POSTER: afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2511–2513. ACM, 2017.

58 Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st international conference on software engineering*, pages 199–209. IEEE, 2009.

59 KissPeter. APIFuzzer. `https://github.com/KissPeter/APIFuzzer`. Accessed: 04-07-2019.

60 Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for javascript object-manipulating programs. *Software: Practice and Experience*, 49(5):840–884, 2019.

61 Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for JavaScript library analysis. In *Proceedings of the 41st International Conference on Software Engineering*, pages 83–93. IEEE Press, 2019.

62 Jonathan Leitschuh. Remote Code Execution — mongo-express. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10758`.

63 Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

64 Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

65 Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

66 Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

67 Guangcheng Liang, Lejian Liao, Xin Xu, Jianguang Du, Guoqiang Li, and Henglong Zhao. Effective fuzzing based on dynamic taint analysis. In *2013 Ninth International Conference on Computational Intelligence and Security*, pages 615–619. IEEE, 2013.

68 V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

69 Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 196–199, 2017.

70 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

71 Trong Nhan Mai. Denial of Service (DoS) — mongo-express. `https://snyk.io/vuln/SNYK-JS-MONGOEXPRESS-1085403`.

72 Michael C Martin and Monica S Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *USENIX Security symposium*, pages 31–44, 2008.

73 Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE, 2008.

74 Amin Milani Fard and Ali Mesbah. Feedback-directed exploration of web applications to derive test models. In *ISSRE*, 2013.

75 Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of node. js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 455–465, 2019.

76 Andres Ojamaa and Karl Düüna. Assessing the security of node. js platform. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 348–355. IEEE, 2012.

77 Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: coverage-guided property-based testing in java. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 398–401. ACM, 2019.

78 Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Dexterjs: robust testing platform for dom-based xss vulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 946–949, 2015.

79 Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *RAID*, 2015.

80 Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168, 2017.

81 Brian Pfretzschner and Lotfi ben Othmane. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–6, 2017.

82 Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 543–553, 2016.

83 Qualys. Web application scanning. `https://www.qualys.com/apps/web-app-scanning/`. Accessed: 04-07-2019.

84 Sazzadur Rahaman, Gang Wang, and Danfeng Yao. Security certification in payment card industry: Testbeds, measurements, and recommendations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 481–498, 2019.

85 Rapid7. Swagger Utility. `https://appspider.help.rapid7.com/docs/swagger-utility`. Accessed: 04-07-2019.

86 Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

87 Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 25–34. IEEE, 2001.

88 Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

89 Max Schlüter. Server crash on POST request. `https://github.com/apostrophecms/apostrophe/issues/1683`.

90 Martin Schneider, Jürgen Großmann, Ina Schieferdecker, and Andrej Pietschker. Online model-based behavioral fuzzing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 469–475. IEEE, 2013.

91 Martin Schneider, Jürgen Großmann, Nikolay Tcholtchev, Ina Schieferdecker, and Andrej Pietschker. Behavioral fuzzing operators for uml sequence diagrams. In *International Workshop on System Analysis and Modeling*, pages 88–104. Springer, 2012.

92 Ben Selwyn-Smith. Remote Code Execution — mongodb-query-parser. `https://www.npmjs.com/advisories/1448`.

93 SmartBear. Openapi specification (fka swagger restful api documentation specification). `https://swagger.io/specification/v2/`. Accessed: 04-07-2019.

94 Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *NDSS*, 2018.

95 Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

96 Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *Acm Sigplan Notices*, 41(1):372–382, 2006.

97 Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.

98 TeeBytes. TnT-Fuzzer. `https://github.com/Teebytes/TnT-Fuzzer`. Accessed: 04-07-2019.

99 Mike Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *2009 30th IEEE symposium on security and privacy*, pages 331–346. IEEE, 2009.

100 Alexandre Teyar. Swurg. `https://github.com/portswigger/openapi-parser`. Accessed04-07-2019.

101 John Toman and Dan Grossman. Concerto: a framework for combined concrete and abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3(POPL):43, 2019.

102 Paolo Tonella and Filippo Ricca. Dynamic model extraction and statistical analysis of web applications. In *Proceedings. Fourth International Workshop on Web Site Evolution*, pages 43–52. IEEE, 2002.

103 Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

104 Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):1–28, 2011.

105 Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, 2008.

106 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

107 Michal Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2015.