SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript

Masudul H. M. Bhuiyan
CISPA Helmholtz Center for Information Security
masudul.bhuiyan@cispa.de

Adithya Srinivas Parthasarathy
IIITDM Kancheepuram
adithyasrinivas11@gmail.com

Nikos Vasilakis
Brown University
nikos@vasilak.is

Michael Pradel
University of Stuttgart
michael@binaervarianz.de

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
staicu@cispa.de

Abstract—**Npm is the largest software ecosystem in the world, offering millions of free, reusable packages. In recent years, various security threats to packages published on npm have been reported, including vulnerabilities that affect millions of users. To continuously improve techniques for detecting vulnerabilities and mitigating attacks that exploit them, a reusable benchmark of vulnerabilities would be highly desirable. Ideally, such a benchmark should be realistic, come with executable exploits, and include fixes of vulnerabilities. Unfortunately, there currently is no such benchmark, forcing researchers to repeatedly develop their own evaluation datasets and making it difficult to compare techniques with each other. This paper presents SecBench.js, the first comprehensive benchmark suite of vulnerabilities and executable exploits for npm. The benchmark comprises 600 vulnerabilities, which cover the five most common vulnerability classes for server-side JavaScript. Each vulnerability comes with a payload that exploits the vulnerability and an oracle that validates successful exploitation. SecBench.js enables various applications, of which we explore three in this paper: (i) cross-checking SecBench.js against public security advisories reveals 168 vulnerable versions in 19 packages that are mislabeled in the advisories; (ii) applying simple code transformations to the exploits in our suite helps identify flawed fixes of vulnerabilities; (iii) dynamically analyzing calls to common sink APIs, e.g., `exec()`, yields a ground truth of code locations for evaluating vulnerability detectors. Beyond providing a reusable benchmark to the community, our work identified 20 zero-day vulnerabilities, most of which are already acknowledged by practitioners.**

I. Introduction

JavaScript is one of the most popular programming languages, and its continuous growth is supported by the npm ecosystem, a repository with more than two million reusable packages. Unfortunately, vulnerabilities are common in npm and pose a major threat to applications and the ecosystem as a whole. A large body of recent research is devoted to the security of the npm ecosystem in general, and server-side JavaScript in particular. Many papers study specific classes of security problems, e.g., code injection [1], [2], ReDoS [3], [4], prototype pollution [5], hidden property attacks [6], path traversal [7], supply chain attacks [8]–[10], outdated [9], [11], [12] or trivial [13] dependencies, and bugs in low-level code [14]. There also is work on developing new defenses and detection techniques using static [2], [5], [10], [15]–[18], dynamic [19]–[24], or hybrid [1], [8] analyses.

Progress in a research community often gets fueled by a reusable benchmark. For example, such benchmarks serve to assess the relative merits of different contributions in databases [25], parallel processing [26], graphics [27], machine learning [28], and bug detection and repair [29].

Unfortunately, despite the importance of server-side JavaScript security, there currently is no comprehensive benchmark of vulnerabilities. Instead, researchers often rely on ad hoc sets of programs collected manually by the authors of the respective papers. While benchmark sets are not without limitations, the lack of a security-oriented benchmark suite for npm has important implications: (1) it slows down research, as authors of each paper have to look for a new set of benchmarks that satisfies their needs and may convince reviewers, and (2) it obfuscates the true merits of different techniques, as they are not compared across a single set of vulnerabilities. A reusable benchmark suite of npm vulnerabilities would alleviate these problems, and help improve experimental techniques and metrics in this area. Specifically, a successful benchmark should provide the following properties:

- *Realistic*: We aim at a benchmark suite built from a diverse set of real-world software, with as few modifications of the original code as possible. Such realism ensures that success on the benchmark is likely to generalize to other real-world security problems.
- *Executable*: We want the suite to include inputs that trigger the vulnerabilities and a runtime oracle that checks whether an attempt to exploit a vulnerability indeed triggers the effect anticipated by the attacker. Having such executable exploits serves as evidence that a vulnerability can be exploited, enables evaluating runtime detection and mitigation techniques, and allows for studying the runtime properties of vulnerabilities.
- *Two-sided*: We aim at a benchmark that includes both vulnerable and fixed versions of the code. Providing both sides is important for measuring the false positive rate of detection and mitigation techniques, but also for studying how vulnerabilities get fixed.

TABLE I
COMPARISON WITH EXISTING VULNERABILITY BENCHMARKS AND DATASETS.

Benchmark/dataset	Language	Vulnerabilities	Realism	Executable exploits	Two-sided	Vetted
CGC [30]	C	590	✗	✓	✗	✓
Juliet [31]	C/C++, Java, C#	121,922	✗	✓	✓	✓
LAVA-M [32]	C	2,265	✗	✓	✓	✗
BigVul [34]	C/C++	3,745	✓	✗	✓	✗
Ferenc et al. [33]	JavaScript	1,496	✓	✗	✓	✗
VulinOSS [39]	various	17,738	✓	✗	✗	✗
Magma [36]	C	118	✓	✗	✓	✓
Ghera [40]	Java/Android	25	✓	✓	✗	✓
Ponta et al. [37]	Java	624	✓	✗	✓	✓
SECBENCH.JS	JavaScript	600	✓	✓	✓	✓

- *Vetted*: It is important to study each vulnerability in the benchmark suite to confirm its existence, ensure that it falls under a certain attack class, and generate appropriate metadata. This requirement is in contrast to large-scale, automatically gathered datasets, which are used, e.g., to train deep learning-based vulnerability detectors, and typically suffer from some noise.

To the best of our knowledge, there currently is no benchmark of vulnerabilities that matches all four of the above criteria. Table I summarizes the most closely related existing benchmarks and datasets. One group of them consists of synthetic vulnerabilities, either created manually [30], via template-based code generation [31], or automatically by mutating existing code [32]. These benchmarks lack realism and hence do not suit our needs. Another group consists of automatically gathered datasets, e.g., created based on commit messages that mention a vulnerability [33], [34]. These datasets do not come with executable exploits and lack manual vetting, which causes noise, e.g., in the form of unrelated code changes tangled with a commit [35]. Finally, the third group of benchmarks consists of manually curated vulnerabilities [36], [37], but lack executable exploits that use the vulnerabilities to trigger a security-relevant action. For example, MAGMA [36] provides inputs that show that vulnerabilities can cause a crash, but not that they can be further exploited by an attacker. Beyond the benchmarks and datasets listed in Table I, there are collections of general bugs [29], [38], which are inspiring but do not focus on security-related problems.

The SECBENCH.JS benchmark suite: This paper presents SECBENCH.JS, the first benchmark suite of JavaScript vulnerabilities that fulfills all four of the above requirements. The benchmark consists of 600 publicly reported vulnerabilities contained in widely-used advisory databases, such as Snyk and GitHub Advisories. To ensure realism, all vulnerabilities are included as-is, without any simplifications of the vulnerable packages. SECBENCH.JS spans the five most common threat classes in the considered databases for benign, server-side JavaScript: path traversal, prototype pollution, command injection, denial-of-service, and code injection. Each vulnerability in SECBENCH.JS includes an executable exploit, i.e., an attack input that triggers the vulnerability and causes behavior unexpected by the vulnerable software package. Additionally, we also provide test oracles to judge the success of each exploit. The benchmark is two-sided: if available, it includes a fix of the vulnerability where the provided exploit often fails. Finally, the benchmark is vetted, as we manually validate each vulnerability.

To showcase the usefulness of SECBENCH.JS, we report several applications of SECBENCH.JS that would either be costly to carry out or error-prone without our suite:

- *Finding mislabeled versions*: By running our exploits on different versions of the vulnerable packages, we identify 168 versions in 19 packages that are incorrectly labeled as not vulnerable in the advisory database. In two cases, the mislabeling even affects the latest release of the package, meaning the packages suffer from zero-day vulnerabilities.
- *Finding flawed patches*: We apply four simple code transformations to the exploits in SECBENCH.JS to test the deployed patches against variants of the original attack. This experiment identifies 18 flawed patches, i.e., new vulnerabilities, which at the time of writing have been assigned 12 CVEs.
- *Dynamically identifying sinks*: Using dynamic analysis, we automatically identify the code locations where the payload provided by an exploit reaches a sensitive API, called *sink*, for 94.5% of the vulnerabilities in our suite. The sink location can be used, e.g., to validate the reports of taint-style, static analyses.

In summary, this paper contributes the following:

- The first benchmark suite of server-side JavaScript vulnerabilities. In contrast to existing benchmarks and datasets, SECBENCH.JS is realistic, provides executable exploits, is two-sided, and has been manually vetted.
- Three concrete applications of the benchmark, which reveal previously unknown vulnerabilities and insights for future work.
- The entire benchmark is publicly available as open-source, well-documented, and packaged conveniently in a container image.

II. METHODOLOGY

In this section, we first discuss our threat model (§II-A) and then present our methodology for building the benchmark suite: collecting known vulnerabilities (§II-B, §II-C), adapting or newly developing proof-of-concept exploits (§II-D), and collecting additional metadata (§II-E).

A. Threat Model

We focus on vulnerabilities, i.e., security defects that are introduced inadvertently by the developers of a package. The reason for this focus is that vulnerabilities account for almost

60% of the problems reported in the considered advisory databases, making them a highly relevant target for a security benchmark. In contrast, we here do not consider malicious packages.

Most of the considered entries in our suite are libraries. By construction, these software components have incomplete threat models: Since it is not clear where the library's inputs come from, it is common to assume the worst-case scenario, i.e., that inputs are attacker-controlled. Our suite does not make any judgment on the feasibility of this threat model, but instead relies on the assessment of practitioners about the risk posed by such vulnerabilities.

We focus on packages that can be executed on the server-side, ignoring typical client-side issues, e.g., cross-site scripting and open redirects. Vulnerabilities in server-side JavaScript are particularly interesting because, unlike on the client side, the code does not run in a sandboxed environment. Instead, a vulnerable package, once exploited, can have serious effects on the broader environment, including reading environment variables, writing files, spawning new processes, and setting up network connections.

To be representative of the threats that practitioners are most interested in, we focus on the largest classes of vulnerabilities present in the advisory databases we consider. Specifically, we select five classes of vulnerabilities, which all are targeted by recent work in this domain: command and code injection [1], [2], [15], [18], [24], [24], ReDoS [3], [4], [19], [20], [41], [42], prototype pollution [5], [15], and path traversal [7], [15].

B. Source of Vulnerabilities

We gather vulnerabilities from three different sources: Snyk, GitHub Advisories (previously called Npm Advisories), and Huntr.dev. These sources include thousands of vulnerabilities, often accompanied by an exploit, are popular among developers, and are extensively used in other work [9], [15], [24], [43]. For each considered source, we scrape its web interface to collect the number of vulnerabilities and their types, which provides the basis for further analysis.

C. Filtering of Candidate Vulnerabilities

We include in our suite only those vulnerabilities for which we can present an input that triggers a security-relevant action, as described below. Thus, we exclude vulnerabilities in packages (a) that we cannot install or that were deleted, (b) that we cannot reproduce, or (c) that are incompatible with our setup, e.g., operating system. When the public vulnerability report contains a proof-of-concept exploit, we adapt this exploit and integrate it into SecBench.js. Otherwise, we try to create an exploit ourselves, allocating a budget of one hour per vulnerability to this task.

D. Executable Exploits

Each exploit in SecBench.js contains five parts: (i) a setup phase in which the target package is imported and initialized, (ii) a sanity check that the security-relevant post-condition is not met before the actual payload, (iii) an API call that delivers the problematic input to the vulnerable API, (iv) a check that the post-condition is met, and (v) a cleanup phase in which the side-effects of the exploit are reverted.

TABLE II
Number of exploits included in SecBench.js, together with the number of metadata entries, for each considered vulnerability type.

Type of vulnerability	**Nb. exploits**	**Fixed version**	**CVE information**
Code injection	40	21	20
Command injection	101	41	90
Path traversal	169	19	80
Prototype pollution	192	126	158
ReDoS	98	78	59
Total	600	285	407

For example, the security-relevant action for injection vulnerabilities is to create a file on disk. As a sanity check, we verify that this file is not yet present on the disk at the beginning of the exploit. After the payload's execution, we assert the presence of the file. Thus, the payload needs to inject code that loads the file system API and creates the target file.

E. Patches of Vulnerabilities

Including the information about the patched version and the fixing commit in our suite is important for enabling studies of deployed fixes, as discussed in Section IV-B. Moreover, having fixed versions of the vulnerable code is essential for judging the precision of static vulnerability detection tools. Such tools should report a problem in the vulnerable version, but not in the fixed version.

We follow two strategies for extracting fixes of vulnerabilities. First, we scrape the considered sources to extract this information. If an advisory refers to a fix for the vulnerability, e.g., in the form of a pull request that fixes the vulnerability, then we include the fix into SecBench.js. Second, because some vulnerabilities are fixed, but the fix is not listed in the corresponding advisory, we search for a fix in case the first strategy is not successful. To this end, we run our exploit on the latest version of the package and check if it still works. In case the exploit fails, we manually analyze the failing exploit to understand if a fix was deployed. If a fix is present, we localize the code location of the fix and then analyze the repository's history to identify the fixing commit.

III. The SecBench.js Benchmark Suite

This section describes the details of SecBench.js, including what vulnerabilities it is composed of (§III-A), how the executable exploits ensure successful exploitations (§III-B), the implementation of the benchmark as a test suite (§III-C), and how the benchmark facilitates deploying program analyses that reason about vulnerabilities (§III-D).

A. Composition of the Benchmark

In total, SecBench.js consists of 600 vulnerabilities, each of which has an executable exploit. Table II how these vulnerabilities are distributed across the five classes of vulnerabilities, and the number of their metadata entries. For 48% of the vulnerabilities, we provide information about the fixed version and the fix commit. In particular, for nine entries, this information was not included in the corresponding security advisory, but we identified it using the second strategy described in Section II-E. We provide CVE information for 67% of the exploits in our suite. The lack of metadata for some entries reflects the limitations of the underlying data, e.g., the fact that for some vulnerabilities, no CVE has been assigned or no patch has been deployed.

384 of the exploits in SECBENCH.JS correspond to vulnerabilities reported both on Snyk and GitHub Security Advisories, while 203 and 7 were exclusively reported on Snyk and GitHub, respectively. For each vulnerability, our metadata links to the corresponding advisories in these databases.

On average, each vulnerable package directly depends on 1.42 other packages, and is depended upon by 947.62 packages. This shows that the considered packages are relatively influential and that there are important interactions with third-party packages.

B. Ensuring Successful Exploitation

To ensure that each exploit in SECBENCH.JS successfully uses the vulnerability to trigger an unforeseen, *security-relevant action*, each of the exploits comes with an *exploit oracle* to check this property. Similar to test oracles, these oracles judge the outcome of an executable exploit via assertions. Unlike regular test oracles, which typically check for desirable behavior, passing the exploit oracle means that the package is indeed vulnerable and that the exploit is successful at abusing the vulnerability.

For most entries in SECBENCH.JS, we create exploits based on existing information, i.e., either a proof-of-concept (PoC) or a natural language description of a possible exploit given in the vulnerability database. A significant amount of work went into encoding this information in a uniform way, adapting it to our framework, and adding exploit oracles. As a measure to approximate this effort, we report the number of exploit assertions, which are all manually created by us: SECBENCH.JS has a total of 1,244 assertions, thus, an average of 2.07 assertions per exploit. Additionally, for most exploits, we have a meta assertion on the number of oracle checks that must be performed during the execution. We found this to be useful for ensuring that all the asynchronous tasks are correctly executed.

An important observation is that exploit oracles and security-relevant actions are specific to a given class of vulnerabilities. For example, one can use a ReDoS vulnerability to trigger a slow computation in the main thread, or a prototype pollution to pollute the global scope. However, since these undesired actions are very different in nature, it is only natural that the oracles that assert their success are also different. The following discusses in detail each type of exploit oracle used by SECBENCH.JS.

- *Code and command injections:* The oracle checks the existence of a custom-named file on the disk. For most exploits, the security-relevant action uses the built-in `fs` module (code injections) or the `touch` UNIX utility (command injection) to create the file. At the beginning of an exploit, the oracle checks that the file is not present on the disk, and after the payload is executed, it asserts its existence.
- *Path traversal:* SECBENCH.JS ships with a file called `flag.txt`, which the path traversal vulnerabilities aim to read. The oracle checks that the content served by the vulnerable npm package is the same as the one in this file. As the absolute path to this file varies from machine to machine, SECBENCH.JS dynamically adjusts the payload to point to the indicated file.
- *Prototype pollution:* The oracle checks the existence of a variable called `polluted` in the global scope. At the beginning of an exploit, the oracle checks that such a value is not present, and once the payload has been executed, it asserts its existence. We note that this is a very strict oracle and that there are prototype pollution attack vectors that do not allow such an action to be performed, e.g., because they only allow for adding properties to specific built-in APIs, such as `Function.prototype`. Since the security impact of such pollutions is limited, we focus on a powerful oracle that checks for the most serious version of these attacks.
- *ReDoS:* The oracle checks that the target payload takes more than a configurable duration d to execute, with $d = 1$ second as the default. While this simple oracle may, in principle, lead to both false negatives and false positives, we minimize their likelihood by (i) dimensioning our exploits so that in our setup the computation takes significantly longer than d, and (ii) the duration d is long enough so that benign computations rarely trigger such long operations.

C. Implementation

At a high level, SECBENCH.JS is a set of unit tests, where each successful exploit is considered a passing test. We use the Jest[1] testing framework, the most popular JavaScript testing framework[2]. SECBENCH.JS relies on package managers, such as `npm` or `yarn`, to distribute the vulnerable package versions. Each vulnerability is contained in a separate folder containing a metadata file with information about the vulnerable version, the fix, the exact location, and external resources, as well as a JavaScript test file with the exploit and the oracle checks. The folder-based structure allows for multiple vulnerable versions of the same package. For example, for `lodash`, SECBENCH.JS includes four different prototype pollution vulnerabilities and one ReDoS vulnerability.

In Figure 1, we present the metadata entry for the vulnerability CVE-2019-10744, an example entry in our suite. Clients of the suite can install the vulnerable version `4.17.10`, e.g., by running `npm install` in the folder containing this file.

[1]https://jestjs.io/
[2]https://2020.stateofjs.com/en-US/technologies/testing/

```
{
  "id": "CVE-2019-10744",
  "dependencies": {
    "lodash":"4.17.10"
  },
  "links": {
    "source1":"SNYK-JS-LODASH-450202",
    "source2":"GHSA-jf85-cpcp-j695"
  },
  "fixedVersion":"4.17.12",
  "fixCommit":"
      c3fd203b3be87a8177f7f00824033c95f981f984",
  "sinkLocation": "lodash.js:2573:21"
}
```

Fig. 1. Metadata corresponding to the entry for lodash's vulnerability CVE-2019-10744. Links are abbreviated for brevity.

```
test("prototype pollution in lodash", () => {
  // setup
  const mergeF = require("lodash").defaultsDeep;
  const payload = '{"constructor": {"prototype":
      {"polluted": "yes"}}}';
  // sanity check
  expect({}.polluted).toBe(undefined);
  // exploit
  mergeF({}, JSON.parse(payload));
  // oracle check
  expect({}.polluted).toBe("yes");
  // cleanup
  delete Object.prototype.polluted;
});
```

Fig. 2. Exploit for `lodash`'s vulnerability CVE-2019-10744.

Figure 2 shows the test that implements the exploit. It first sets up the test by importing the vulnerable package and initializing relevant constants, and then asserts that the target property is not inadvertently present in the global scope. After that, the test triggers the payload by passing it to the vulnerable module. Finally, the test assesses the presence of the target property and hence, the exploit's success. Some tests in SecBench.js have more complex setup and teardown phases. For example, for path traversal vulnerabilities, we often need to start the target package in a separate process to act as a web server, perform an HTTP request, and finally, stop the server. Nonetheless, the high-level structure of all our tests is similar to the one discussed above.

To simplify batch installation, SecBench.js organizes its metadata and entries in a hierarchical structure. At first, we aggregate the individual vulnerability folders by vulnerability class, offering an aggregated `package.json` file that refers to all the individual vulnerabilities. For example, all ReDoS vulnerabilities are comprised in the `redos` folder, containing a `package.json` with 98 internal dependencies. Running the package manager in that folder will download all the 98 vulnerabilities of that class, together with their dependencies. Similarly, SecBench.js also allows for batch installing all the 600 vulnerable packages, by providing a main `package.json` that refers to the individual ones corresponding to the five vulnerability classes considered in the benchmark. On a standard laptop, it takes about twelve minutes to run `npm install` in the main folder of SecBench.js, i.e., installing all the 600 vulnerable packages, with their exploits.

Once installed, a user can run all tests sequentially or in parallel. By default, `jest` runs all the tests in parallel, achieving the following test execution times on a standard laptop: 57s, 13s, 518s, 12s, and 156s, for code injection, command injection, path traversal, prototype pollution, and ReDoS, respectively. The two slowest classes are ReDoS, for which the payloads trigger a significant slowdown in the target packages, and path traversal, for which we run the tests sequentially because multiple of them try to serve requests on the same HTTP port.

D. Deploying Analysis Code

An important use case we envision for our suite is the development of runtime analyses to detect and defend against exploits. To this end, one needs to deploy analysis code that runs along with our exploits. We outline below two ways in which analysis code can be injected. First, due to the integration between Jest and Babel[3], an off-the-shelf transpiler for JavaScript, one can deploy runtime instrumentation using Babel. Second, a widely-used dynamic analysis technique for scripting languages is monkey patching, i.e., augmenting the semantics of built-in APIs to include analysis code. By leveraging Jest's setup phase, one can define custom analysis files that are executed before each test and can thus alter the test's global scope. We successfully use this analysis technique to extract the sink location for the vulnerabilities in our suite (§IV-C).

IV. Applications of SecBench.js

The availability of SecBench.js enables studying properties of vulnerabilities via dynamic analysis at a previously impossible scale. The following shows three examples of such studies. We first study how many versions of a given package are affected by our exploits, and whether this information matches the version constraints provided in the advisory (Section IV-A). Then, we show that applying simple mutations to our exploits helps identify zero-day vulnerabilities in insufficiently patched packages (Section IV-B). Finally, we present a dynamic analysis to precisely identify the location where an attacker-provided value gets passed to a sensitive API (Section IV-C).

A. Finding Mislabeled Vulnerable Versions

Vulnerability reports often indicate the versions of a package that are affected by a vulnerability, which is useful for guiding clients of the package to upgrade their dependencies. Since these reports are manually curated, they may be inaccurate, e.g., due to regressions and flawed fixes. The exploits provided in SecBench.js allow for automatically checking which versions of a package can be successfully exploited. The following shows how cross-checking this information against

[3]https://babeljs.io/

the version ranges reported to be vulnerable helps to find mislabeled vulnerable versions.

a) Experimental Setup: For every package in SecBench.js, we iteratively update it to every one of its versions and attempt to run the exploit against that particular version, storing the test outcome for each attempt. Because running an exploit on all versions of a package is non-trivial, this setup yields an underapproximation of versions that are affected by an exploit. For example, legacy versions of a package may not be compatible with our setup, or there may be breaking changes of the used APIs, along the history of the project. Finally, we check if all the vulnerable versions, as identified with SecBench.js, are flagged accordingly by the corresponding vulnerability database.

b) Results: Most of the considered packages have less than 20 versions, while there are a handful of packages with thousand of versions. Figure 3 shows how many of these versions are found to be vulnerable with the exploits in SecBench.js. The mean number of vulnerable versions per package is 19.1, with 50% of the packages having ≤ 5 vulnerable versions, while `@firebase/util` has a maximum of 1,487 vulnerable versions. Overall, the figure shows that most vulnerabilities affect multiple versions, sometimes even more than 100.

For each package in SecBench.js, we check if there are inconsistencies between the vulnerable versions we detected and the original advisory. We manually check every mismatch and confirm 168 vulnerable versions in 19 packages that are incorrectly flagged as not vulnerable by Snyk. We are in the process of reporting these inconsistencies with associated examples to Snyk.

c) Examples: In Figure 4, we show examples of mismatches between SecBench.js and the Snyk database. As mentioned earlier, our analysis might produce false negatives; thus, we only focus on cases where the analysis flagged a version as vulnerable, but the public vulnerability report claims the version to not suffer from the vulnerability. We discover three classes of mismatches. First, there are cases like `changeset` or `underscore.string`, where the constraint provided in the security advisory fails to capture legacy versions, released before the actual interval specified by the constraint. Second, there are cases like `ts-dot-prop` when the constraint fails to capture versions after the indicated interval. Such errors might give users a false sense of security, since audit tools like `snyk-cli` would flag these versions as safe to use. Nonetheless, the affected releases might still be considered as legacy. Finally, for two packages, the latest version of the package is incorrectly flagged as not vulnerable, hence, representing zero-day vulnerabilities. Since developers are usually encouraged to migrate whenever possible to the latest version, having these versions incorrectly flagged as not vulnerable anymore is a severe problem and gives a false sense of security.

For example, the initial security advisory for `jspdf` identifies the following regular expression vulnerable to ReDoS, in version 2.3.0 and earlier:

```
/^data:(\w*\/\w*);*(charset=[\w=-]*)*;*$/
```

The regular expression consists of a wild card in the second capturing group `(charset=[/\w=-]*)*` which triggers the so-called "catastrophic backtracking". Since there was no exploit available for this vulnerability, we developed our own by studying the original vulnerable expression and found the following input to trigger the problem:

```
data:/charset=charset=charset=charset=charset=
    charset=charset=charset=charset=charset=
    charset=charset=charset=!
```

The advisory indicates that the problem was fixed in a later version and links to a commit that refines the regular expression into the following:

```
/^data:(\w*\/\w*);*(charset=(?!charset=)[\w=-]*)
    *;*$/
```

The look-ahead is supposedly preventing the catastrophic backtracking. However, because the patch uses a negative look-ahead to match the character literals $charset =$, it is still possible to perform a ReDoS attack, e.g., using the following input:

```
data:image/jpeg;charset=xcharset=xcharset=xcharset
    =xcharset=xcharset=xcharset=xcharset=xcharset=
    xcharset=xcharset=xcharset=xcharset=xcharset=
    xcharset=xcharset=xcharset=xcharset=xcharset=
    xcharset=xcharset=xcharset=xcharset=xcharset=
    xcharset=x!base64
```

As another interesting example, consider the case of the `mithril` package shown in Figure 5. This package has three major releases, all of which were concurrently maintained for some time. Due to this parallel maintenance, the natural ordering of versions does not match with the publishing order. By natural order, we mean that a version n.0.0 gets published before (n+1).0.0 or n.1.0, and the later version would be considered the latest version. But in the case of `mithril`, version 1.1.7 was published on September 23, 2019, i.e., after version 2.0.4, which was published on August 18, 2019, and marked as latest on npm.

These cases of parallel versioning make it difficult to correctly specify the constraint that indicates vulnerable versions. To specify the vulnerable versions for `mithril`, the Snyk database uses this complex constraint:

$$>= 1.0.0 \ < 1.1.7, >= 2.0.0 \ < 2.0.3$$

However, since versions like 1.1.7 are released later than 2.0.3, it is not trivial for developers to determine whether a given version they are using is vulnerable or not. We identify multiple release candidate versions, e.g., 2.0.0-rc.0 that are actually vulnerable, but that are not captured by the constrained above. As a result of such mislabeling, developers may get a false sense of security and accidentally use vulnerable versions.

d) Implications: The findings in this section show that the ability to cross-check manually specified version ranges against the executable exploits in SecBench.js is useful to

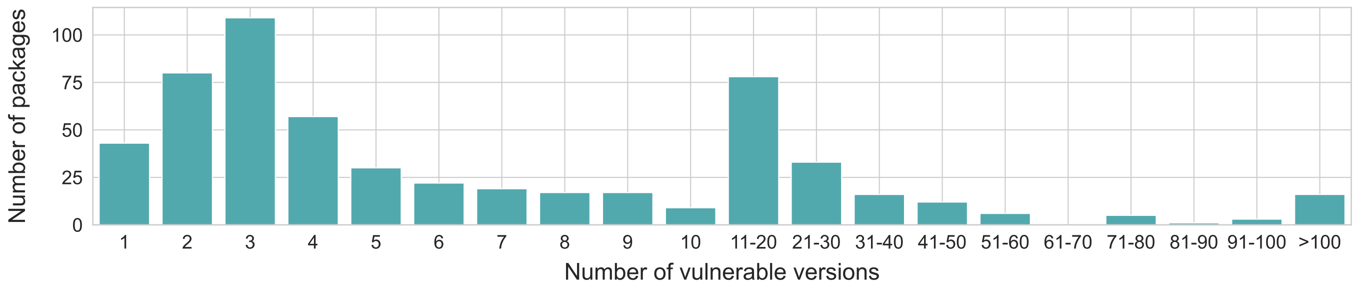

Fig. 3. Distribution of the number of vulnerable versions of a package that are exploitable with SecBench.js.

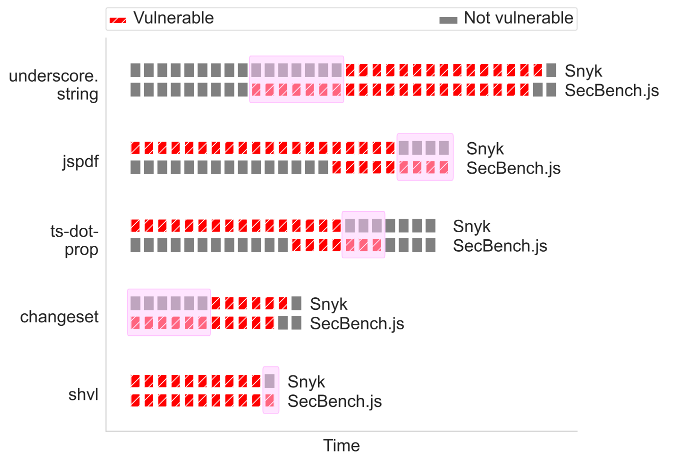

Fig. 4. Examples of disagreements between our dynamic results and the Snyk database. Each box represents a version of the package. For each package, the lower line shows the assessment in SecBench.js, while the upper line shows the available vulnerable version information in the Snyk database. The two lines are synchronized in the sense that two boxes corresponding to the same x-point depict the same package version. The pink overlay points to versions that are falsely labeled as non-vulnerable in the Snyk database.

detect mislabeled versions. We also find that accurately specifying which versions are affected by a vulnerability is non-trivial, e.g., due to multiple major versions being maintained in parallel.

B. Finding Flawed Fixes

When addressing a vulnerability, developers may sometimes overfit to a proof-of-concept provided in the original advisory. As a result, the published fix may not fully address the problem, leaving some attack vectors open to be exploited. For example, for a prototype pollution vulnerability, one can pollute the global object using both the paths `obj.__proto__` and `obj.constructor.prototype`. Hence, a fix that considers only one of these paths would be flawed.

a) Experimental Setup: To detect flawed fixes based on SecBench.js, we design three simple mutations, shown in the first column of Table III. The last two capture the case described earlier, while the first mutation corresponds to a type confusion problem[4]. We then update all the vulnerable packages in SecBench.js to their latest versions and only consider those packages for which the original exploit does not work, i.e., they are supposedly fixed. We then apply each of the three mutations above to the exploit in SecBench.js and rerun the modified exploit. If the test succeeds, we have identified a flawed fix for the corresponding package.

b) Results: In total, we find thirteen, four, and one zero-day vulnerabilities for the three mutations, respectively. We reported all these issues to the maintainers and, until the time of writing, we got assigned twelve new CVEs for our findings, as shown in Table III. For two cases, there was a concurrent disclosure pending for the same issue, and the rest are still in the disclosure process.

c) Examples: Let us consider the case of `convict`, a popular package from Mozilla for managing configuration files. In response to the original vulnerability report for this package, the authors deployed a fix[5] in the `set` method, by including the `if` statement below:

```
const path = k.split('.')
const childKey = path.pop()
const pKey = path.join('.')
if (!(pKey == '__proto__' || pKey == 'constructor'
     || pKey == 'prototype')) {
 const parent = walk(this._instance, pKey, true)
 parent[childKey] = v
}
```

The check prevents writing paths like `"__proto__.x"`, but not composed ones like `"constructor.prototype.x"`, where `pKey` is assigned `"constructor.prototype"`. In response to our report, the maintainers deployed a more sophisticated fix that also considers this additional attack vector.

d) Implications: An alternative to the experiments described in this section would be to manually analyze all the deployed patches for the 192 prototype pollutions vulnerabilities in our suite. While this is doable, the effort would be considerably higher than writing the simple mutations in Table II and running the suite three times. Moreover, we can run such experiments on a regular basis, for all future versions of the target packages, to detect possible regressions. Thus, we conclude that an executable vulnerability database, such

[4]https://snyk.io/blog/remediate-javascript-type-confusion-bypassed-input-validation

[5]https://github.com/mozilla/node-convict/commit/688c46afe099b44512665dee6263eacd9f4f71a8

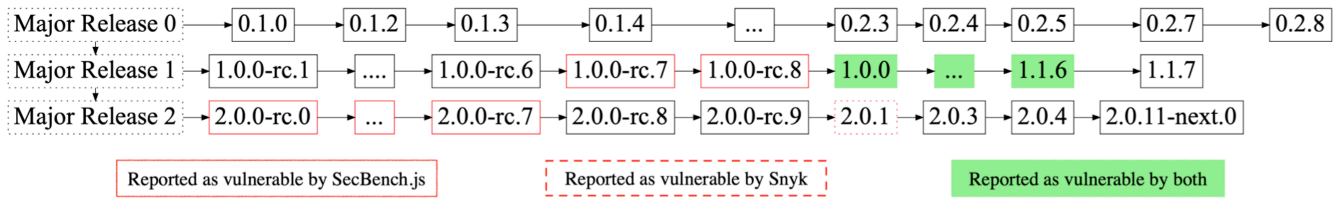

Fig. 5. Example of a disagreement between our dynamic results and the Snyk database in the `mithril` package.

TABLE III
MUTATIONS FOR IDENTIFYING PROBLEMATIC FIXES, AND THE ZERO-DAY VULNERABILITIES IDENTIFIED BY EACH MUTATION.

Mutation	**Security advisories**
`"__proto__"` → `["__proto__"]`	CVE-2021-23518, CVE-2021-23760, CVE-2021-23507 CVE-2021-23497, CVE-2021-23460, CVE-2021-23558 CVE-2022-25354, CVE-2022-25296, CVE-2022-25352
`"__proto__"` → `"constructor.prototype"`	CVE-2022-22143, CVE-2022-24279
`"__proto__": {...}` → `"constructor": {"prototype": {...}}`	CVE-2021-23470

as SECBENCH.JS, is useful for validating deployed fixes and for identifying regressions.

C. Localizing Sink Calls

Most vulnerabilities can be described as a taint analysis-style flow of data from a source to a sink. A sink here is a built-in API that is used to deliver the payload. For example, a typical sink for command injection is `child_process.exec()`. Having precise information about the sink location is useful to better understand a vulnerability. Moreover, knowing the sink location provides a ground truth for evaluating taint-style analyses that report potential vulnerabilities. The following shows how to use SECBENCH.JS to localize sink calls via dynamic analysis.

a) Experimental Setup: To extract sink locations, we use a simple yet effective dynamic analysis. We run each exploit in a prepared environment in which relevant sink APIs are hooked. For example, for prototype pollution we add a custom setter on the property `polluted` in `Object.prototype`, i.e., the property that our payloads are trying to set. Each time an exploit sets that property, our analysis code is triggered, and we inspect the stack trace to extract the location that triggers the property set. Similarly, we hook `child_process` APIs for command injection, `fs` for path traversal, regular expression matching for ReDoS, and `eval` and `Function` for code injection. In case of multiple sink calls, we only extract the first sink location.

b) Results: Applying the dynamic analysis to all vulnerabilities in SECBENCH.JS finds a sink location for 567 of 600 exploits (94.5%). The remaining cases are missed due to complex behavior calls not captured by our simple dynamic analysis. For example, for path traversal exploits, the main process creates a new detached process, where the actual sink calls happen. However, due to the newly created process, the dynamic analysis fails to trace the sink calls in a few cases.

c) Implications: Having a large set of precisely identified sink locations provides a basis for evaluating existing and future taint-style analyses. Such analyses, both static and dynamic, typically warn users about unexpected flows from an API entry point to a sink location. The sink locations obtained using SECBENCH.JS will serve as a ground truth for evaluating the results of taint-style analyses.

V. DISCUSSION

This section discusses the application of SECBENCH.JS in practice, its relationship with other language ecosystems, and some of its key limitations.

a) Applying SECBENCH.JS: We see several opportunities for applying SECBENCH.JS in future security research. First, SECBENCH.JS is well-suited for evaluating mitigation techniques: a plethora of security systems [1], [19], [23], [24] aims to mitigate or eliminate unintended behaviors during the execution of a program. SECBENCH.JS can be used to empirically characterize the success of these systems, including both mechanisms and policies safeguarding program execution. SECBENCH.JS's ability to trigger vulnerabilities and confirm the anticipated side-effects via runtime oracles is critical here, including the decision to offer automation and minimize other side-effects that could interfere with mitigation techniques.

Second, SECBENCH.JS can be used to evaluate both static and dynamic vulnerability detection techniques, i.e., whether a tool or a system designed to infer legitimate behavior or detect certain classes of attacks indeed succeeds in such inference or detection. The existence of both vulnerable and non-vulnerable versions of the code in SECBENCH.JS is important for this goal, as they can be used to characterize precision and recall.

Finally, SECBENCH.JS can also be used for in-depth studies and analyses of real-world vulnerabilities and the features that enable such vulnerabilities in practice, e.g., code and object complexity or source-target distance in the object graph.

b) Relation to other languages and ecosystems: The choice of a particular language and runtime environment—server-side JavaScript and Node.js—necessarily affects the classes of threats that are part of a benchmark suite. Server-side JavaScript means that a vulnerable component does not run in the sandboxed environment of a web browser. Once exploited, a vulnerable package can thus have serious effects on the broader environment in which the program is executing, including reading environment variables, writing files, spawning new processes, and setting up network connections. Many of these threat classes are common in other languages and ecosystems, such as Python (PyPI), Ruby (Rubygems), and Java (Maven Central).

A few threats that are part of SecBench.js are not commonly found in other environments and are due to design decisions related to the semantics and implementation of JavaScript. For example, prototype pollution attacks are due to the combination of mutable intrinsics and runtime resolution available in the JavaScript language. As another example, ReDoS attacks are due to cooperative task scheduling present in the JavaScript runtime environment. These behaviors exist in other environments (mutable intrinsics in Smalltalk; cooperative scheduling in Lua) but have not received the attention they have received in JavaScript.

At the same time, some other classes of vulnerabilities that are possible or even common in other environments are not part of SecBench.js. One example is vulnerabilities stemming from the lack of memory safety, common in components developed in memory-unsafe languages, such as C and C++. Another example is cross-site scripting attacks, common in components targeting front-end web applications.

VI. Threats to Validity

SecBench.js is a vetted benchmark suite, i.e., all vulnerabilities have been manually inspected to validate their existence and that their associated exploits and metadata are proper. Despite our best efforts, the results of this manual inspection may nevertheless be subject to occasional errors. The validity of any conclusions draw based on SecBench.js are limited to the packages, vulnerabilities, exploits, and classes of threats included in the suite. In particular, this means that using the suite does not allow to generalize results to other programming languages or environments. Given the importance of server-side JavaScript, especially from a security point of view, we consider SecBench.js to nevertheless offer a valuable contribution.

SecBench.js is intended for research and thus provides infrastructure to automate, instrument, and validate the execution of the exploits included in SecBench.js. There is a small chance that this infrastructure might accidentally interfere with security policies or mechanisms associated with the artifact being evaluated. For example, SecBench.js's creation of new processes might interfere with systems that detect or mitigate process-related limits; and its execution in a dedicated container environment might interfere with policies related to limitation and prioritization of operating-system resources. Some of these limitations would be present in any evaluation infrastructure, i.e., even with ad-hoc benchmarks, i.e., while others can be ameliorated through careful engineering of the artifact under evaluation.

VII. Related work

A. Vulnerability datasets

There are different kinds of vulnerability datasets. One of them is benchmarks of vulnerable programs aimed to be used for evaluating static analyzers or fuzzers [31], [32], [36]. The most closely related such benchmark is Magma [36], which is also built from real-world vulnerabilities and comes with inputs to exploit them, but targets C instead of JavaScript. Moreover, their exploitation is limited to a crash, while our testing oracle assert the success of as security-relevant action. The AEG exploit generation system [44] aims at finding vulnerabilities and generating exploits automatically.

Another class of related work consists of large-scale datasets extracted in an automated manner, e.g., from version histories [33], [34], [39], [45], intended as training data for machine learning-based vulnerability detection. Due to their automated creation, these datasets do not come with exploits and suffer from some degree of noise (e.g., 53% true positives based on manual inspection [45]).

Finally, a third kind of dataset offers manually validated vulnerabilities and exploits for them [37], [40], similar to SecBench.js, but none of them targets JavaScript. Beyond vulnerabilities, other benchmark suites [26], [46], [47] are mainly designed to study a specific program area outside software security. To the best our knowledge, we are the first to construct a benchmark of executable, real-world vulnerabilities in JavaScript code.

B. Bug benchmarks

Looking beyond the security domain, there are various benchmarks of general bugs, such as Defects4J [29], Bugs.jar [48], BugSwarm [38] for Java, BugsJS for JavaScript [49], and a set of JavaScript performance bugs [50]. Other benchmarks focus on concurrency bugs [51], [52], high-impact bugs [53], and non-functional bugs [54]. While many of these benchmarks also provide inputs to trigger the bugs, they do not focus on vulnerabilities.

C. npm and other package ecosystems

The prevalence of vulnerabilities in npm and other package ecosystems has motivated various studies and techniques to understand and identify ecosystem-level security issues. Zimmermann et al. [9] study ecosystem-level security threats in npm. Others study the impact of vulnerabilities on package dependency network [43], the phenomenon of "trivial" packages in npm [13], or the impact of ReDoS vulnerabilities [4]. Supply chain attacks are another problem of specific interest [8], [16]. Pashchenko et al. [12] report on an interview-based study to understand the (lack of) dependency management. All the above highlights the importance of security threats in large-scale package ecosystems. SecBench.js will help address these threats by providing a benchmark for evaluating future detection and mitigation tools.

D. *JavaScript security*

There are various techniques for detecting JavaScript vulnerabilities and for mitigating their exploitation, of which we discuss a representative sample. Many techniques detect a particular kind of vulnerability, such as injection vulnerabilities [1], [2], hidden property attacks [55], ReDoS vulnerabilities [3], and prototype polution [5]. Others provide more general detection techniques, e.g., in the form of static extraction of taint specifications [56], dynamic taint analysis [22], and graph-based vulnerability detection [15]. There are mitigations against ReDoS [19]–[21], in the form of compartementalization [23], privilege reduction [24], and debloating of packages [11]. Beyond the JavaScript code itself, security-relevant bugs in the language implementation are another concern [14], [57]. We envision SecBench.js to help compare and improve techniques that detect JavaScript vulnerabilities and that mitigate their exploitation.

VIII. Conclusion

Computer science research and development depend crucially on benchmarks that provide a common foundation for evaluating techniques, systems, and solutions. Security research for server-side JavaScript currently lacks a comprehensive set of real-world executable benchmarks, collected and analyzed systematically. As a result, researchers are forced to evaluate their contributions ad hoc, hampering the direct comparison among different techniques—a problem we have repeatedly faced ourselves and heard from others when developing systems targeting defensive software security. This paper makes a first step towards addressing this problem by introducing SecBench.js, a benchmark suite of vulnerabilities and executable exploits for server-side JavaScript. SecBench.js contains 600 real-world vulnerabilities and executable exploits, spread across several threat classes. SecBench.js is offered in a fully automated package that contains (1) exploit oracles for automatically verifying the success of each exploit, (2) pointers to the fixed versions of each benchmark, and (3) additional metadata, such as contextual information—all enabled by a manual study of each vulnerability in the benchmark set. We perform several experiments to show the usefulness of the suite: identify flawed fixes and inaccuracies in the security advisories, extract vulnerability locations, and identify deployed patches. As a result of these experiments, we uncover 20 zero-day vulnerabilities, for which we were assigned 12 CVEs. We believe that these initial results show the potential of executable vulnerability databases like SecBench.js, and we hope that the community will join in the effort of extending and maintaining this benchmark suite.

Data Availability

The benchmark suite, including all associated code and data, will be available as open-source. While the paper is under review, we provide a snapshot of the repository in HotCRP. The shared data includes a comprehensive README file that describes (i) the structure of the suite and (ii) the most important scripts to be run to install and execute the suite.

References

[1] C. Staicu, M. Pradel, and B. Livshits, "SYNODE: understanding and automatically preventing injection attacks on NODE.JS," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[2] F. Gauthier, B. Hassanshahi, and A. Jordan, "AFFOGATO: runtime detection of injection attacks for node.js," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[3] C. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *USENIX Security Symposium*, 2018.

[4] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2018.

[5] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node.js prototype pollution vulnerabilities via object lookup analysis," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2021.

[6] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the Node.js ecosystem," in *USENIX Security Symposium*, 2021.

[7] L. Gong, "Dynamic analysis for javascript code," Ph.D. dissertation, University of California, Berkeley, 2018.

[8] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[9] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *USENIX Security Symposium*, 2019.

[10] M. Taylor, R. K. Vaidya, D. Davidson, L. D. Carli, and V. Rastogi, "Defending against package typosquatting," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[11] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node. js applications," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[12] I. Pashchenko, D. L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Conference on Computer and Communications Security (CCS)*, 2020.

[13] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[14] F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *Symposium on Security and Privacy (S&P)*, 2017.

[15] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining Node.js vulnerabilities via object dependence graph and query," in *USENIX Security Symposium*, 2022.

[16] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Towards using source code repositories to identify software supply chain attacks," in *Conference on Computer and Communications Security (CCS)*, 2020.

[17] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node.js applications," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[18] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: feedback-driven static analysis of node.js applications," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (FSE)*, 2019.

[19] J. C. Davis, F. Servant, and D. Lee, "Using selective memoization to defeat regular expression denial of service (ReDoS)," in *Symposium on Security and Privacy (S&P)*, 2021.

[20] J. C. Davis, "Rethinking regex engines to address redos," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2019.

[21] J. C. Davis, E. R. Williamson, and D. Lee, "A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning," in *USENIX Security Symposium*, 2018.

[22] R. Karim, F. Tip, A. Sochurkova, and K. Sen, "Platform-independent dynamic taint analysis for JavaScript," *IEEE Transactions on Software Engineering*, 2018.

[23] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *Network and Distributed System Security Symposium, (NDSS)*, 2018.

[24] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing dynamic library compromise on Node.js via RWX-based privilege reduction," in *Conference on Computer and Communications Security (CCS)*, 2021.

[25] M. Pöss and C. Floyd, "New TPC benchmarks for decision support and web commerce," *SIGMOD Rec.*, 2000.

[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[27] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The german traffic sign recognition benchmark: A multi-class classification competition," in *International Joint Conference on Neural Networks (IJCNN)*, 2011.

[28] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "EMNIST: extending MNIST to handwritten letters," in *International Joint Conference on Neural Networks (IJCNN)*, 2017.

[29] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2014.

[30] B. Caswell, "Cyber grand challenge corpus." [Online]. Available: http://www.lungetech.com/cgc-corpus/

[31] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and java test suite," *Computer*, 2012.

[32] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. K. Robertson, F. Ulrich, and R. Whelan, "LAVA: large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[33] R. Ferenc, P. Hegedüs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy, "Challenging machine learning algorithms in predicting vulnerable javascript functions," in *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2019, Montreal, QC, Canada, May 28, 2019*, T. Menzies and B. Turhan, Eds. IEEE / ACM, 2019, pp. 8–14. [Online]. Available: https://doi.org/10.1109/RAISE.2019.00010

[34] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *International Conference on Mining Software Repositories (MSR)*, 2020.

[35] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, T. Zimmermann, M. D. Penta, and S. Kim, Eds. IEEE Computer Society, 2013, pp. 121–130. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624018

[36] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," in *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, L. Huang, A. Gandhi, N. Kiyavash, and J. Wang, Eds., 2021.

[37] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *International Conference on Mining Software Repositories (MSR)*, 2019.

[38] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 339–349. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00048

[39] A. Gkortzis, D. Mitropoulos, and D. Spinellis, "Vulinoss: a dataset of security vulnerabilities in open-source systems," in *International Conference on Mining Software Repositories (MSR)*, 2018.

[40] J. Mitra and V. Ranganath, "Ghera: A repository of android app vulnerability benchmarks," in *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2017.

[41] Y. Liu, M. Zhang, and W. Meng, "Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities," in *Symposium on Security and Privacy (S&P)*, 2021.

[42] Z. Bai, K. Wang, H. Zhu, Y. Cao, and X. Jin, "Runtime recovery of web applications under zero-day redos attacks," in *Symposium on Security and Privacy (S&P)*, 2021.

[43] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *International Conference on Mining Software Repositories (MSR)*, 2018.

[44] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: automatic exploit generation," in *Network and Distributed System Security Symposium (NDSS)*, 2011.

[45] Y. Zheng, S. Pujar, B. L. Lewis, L. Buratti, E. A. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2A: A dataset built for ai-based vulnerability detection methods using differential analysis," in *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021.* IEEE, 2021, pp. 111–120. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP52600.2021.00020

[46] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, 2006.

[47] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: java benchmarking development and analysis," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[48] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: a large-scale, diverse dataset of real-world Java bugs," in *International Conference on Mining Software Repositories (MSR)*, A. Zaidman, Y. Kamei, and E. Hill, Eds., 2018.

[49] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark of javascript bugs," in *Conference on Software Testing, Validation and Verification, (ICST)*, 2019.

[50] M. Selakovic and M. Pradel, "Performance issues and optimizations in JavaScript: An empirical study," in *International Conference on Software Engineering (ICSE)*, 2016, pp. 61–72.

[51] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, "Jacontebe: A benchmark suite of real-world java concurrency bugs," in *International Conference on Automated Software Engineering (ASE)*, 2015.

[52] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, "Gobench: A benchmark suite of real-world go concurrency bugs," in *International Symposium on Code Generation and Optimization, (CGO)*, 2021.

[53] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, "A dataset of high impact bugs: Manually-classified issue reports," in *International Conference on Mining Software Repositories (MSR)*, 2015.

[54] A. Radu and S. Nadi, "A dataset of non-functional bugs," in *International Conference on Mining Software Repositories (MSR)*, 2019.

[55] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the node.js ecosystem," in *USENIX Security Symposium*, 2021.

[56] C. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *International Conference on Software Engineering (ICSE)*, 2020.

[57] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé *et al.*, "Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases," in *Network and Distributed System Security Symposium (NDSS)*, 2021.