PoCGen: Generating Proof-of-Concept Exploits for Vulnerabilities in Npm Packages

Deniz Simsek
University of Stuttgart
Germany
simsekdz@studi.informatik.uni-stuttgart.de

Aryaz Eghbali
University of Stuttgart
Germany
aryaz.eghbali@iste.uni-stuttgart.de

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

***Abstract*—Security vulnerabilities in software packages are a significant concern for developers and users alike. Patching these vulnerabilities in a timely manner is crucial to restoring the integrity and security of software systems. However, previous work has shown that vulnerability reports often lack proof-of-concept (PoC) exploits, which are essential for fixing the vulnerability, testing patches, and avoiding regressions. Creating a PoC exploit is challenging because vulnerability reports are informal and often incomplete, and because it requires a detailed understanding of how inputs passed to potentially vulnerable APIs may reach security-relevant sinks. In this paper, we present PoCGen, a novel approach to autonomously generate and validate PoC exploits for vulnerabilities in npm packages. This is the first fully autonomous approach to use large language models (LLMs) in tandem with static and dynamic analysis techniques for PoC exploit generation. PoCGen leverages an LLM for understanding vulnerability reports, for generating candidate PoC exploits, and for validating and refining them. Our approach successfully generates exploits for 77% of the vulnerabilities in the SecBench.js dataset and 39% in a new, more challenging dataset of 794 recent vulnerabilities. This success rate significantly outperforms a recent baseline (by 45 absolute percentage points), while imposing an average cost of $0.02 per generated exploit.**

***Index Terms*—Vulnerability, Exploit Generation, Large Language Models**

I. Introduction

Security vulnerabilities pose a major threat to software and users alike, with the number of reported vulnerabilities increasing each year. In 2024 alone, over 40,000 CVEs were disclosed, an increase of 38% over the previous year [1]. As software ecosystems become more complex and interdependent, mitigating vulnerabilities becomes increasingly challenging. This holds particularly for Node.js and its package manager, npm, which form the backbone of modern JavaScript and TypeScript development. With millions of packages and a dense network of dependencies, the npm ecosystem is susceptible to a wide range of security risks [2], including transitive vulnerabilities, where a single vulnerable package can propagate security risks across thousands of applications [3].

When a vulnerability is discovered, it is typically reported to the developers of the affected package, who are then expected to create a patch to fix the issue. Once the vulnerable software is fixed, or some time has passed from the initial vulnerability report, the vulnerability report is published as a Common Vulnerabilities and Exposures (CVE) entry. The process of fixing vulnerabilities is often facilitated by a proof-of-concept (PoC) exploit, which demonstrates how the vulnerability can be exploited in practice. Moreover, PoC exploits are useful for testing the patch and preventing regressions in the future. However, many vulnerability reports lack a PoC exploit [4], and even many CVE reports do not have any PoC exploit. For example, in the SecBench.js [5] dataset, only 179 out of 560 CVEs contain a PoC exploit in the report. Furthermore, the publicly available exploits are not reliable, and in some cases are malicious themselves [6].

Fig. 1: CVE-2024-57063 report with no PoC exploit.

```
A prototype pollution in the lib function of
php-date-formatter v1.3.6 allows attackers to cause
a Denial of Service (DoS) via supplying a crafted
payload.
```

As a real-world example, the vulnerability CVE-2024-57063 in the "php-date-formatter" package shown in Fig. 1 does not contain a PoC exploit in the report. PoC exploits are useful for preventing regressions and identifying partial or ineffective fixes that fail to resolve the underlying vulnerability. The process of creating PoC exploits is often time-consuming and requires a deep understanding of the codebase, the vulnerability, and the underlying technology [5]. Particularly in the case of Fig. 1, the vulnerability description does not mention which function is vulnerable, and does not provide any information about the input that triggers the vulnerability.

With LLMs' abilities in understanding and generating source code in addition to natural language, they have shown great promise in various software engineering tasks, including test generation, and program repair [7]–[9]. Hence, LLMs are good candidates to address the challenges in generating PoC exploits from vulnerability reports, specifically when the reports are vague. Moreover, with their understanding of vulnerability types, they can generate payloads for exploits in a more targeted manner, compared to traditional approaches that rely on symbolic execution or fuzzing [10], [11].

Recently, Marques et al. [12] proposed a method to find four vulnerability types in Node.js packages and generate exploits for them in a tool called *Explode.js*. Their approach uses taint analysis, a set of exploit templates, and symbolic execution to trigger vulnerabilities and output the exploits. Their work addresses the problem of generating PoC exploits for vulnerable npm packages. However, their approach does not use the vulnerability report and tries to find the vulnerabilities with taint analysis.

Since the recall of their approach on average is 58.2% and at most 66.2% for command injection vulnerabilities, there are vulnerabilities detected by developers or security researchers that are not detected by their approach. Therefore, generating PoC exploits from natural language description is of value to the community. Moreover, the exploit rate of their approach is on average 46.1% and at most 51.8% for prototype pollution vulnerabilities. Even for the vulnerabilities that are detected by their approach, on average 22% cannot be exploited by Explode.js.

This paper aims to address these limitations by utilizing LLMs and program analysis techniques to automate the generation of PoC exploits from vulnerability reports in npm packages. Our proposed approach, PoCGen, takes as input an informal description of a vulnerability, as found in CVE reports, and automatically generates an executable exploit using a combination of LLM prompting and static and dynamic analysis. PoCGen consists of four, iteratively executed components: (i) understanding the vulnerability and extracting source-level information, (ii) generating the exploit, (iii) validating the exploit, and (iv) refining the prompt. Component (i) uses dynamic analysis to explore the package's exported functions and prompts the LLM with the given vulnerability report to identify the vulnerable function. Component (ii) generates a candidate exploit using an LLM, which receives a prompt that contains a taint path from the vulnerable function to the exploit target. Component (iii) executes and validates the candidate exploit against a test oracle. If the candidate exploit is not valid, PoCGen uses component (iv) and refines the prompt using a set of refiners that provide static or dynamic information to component (ii) where the LLM attempts again to generate a valid exploit. This process continues until either a valid exploit is generated or a predefined budget is exhausted.

To generate a PoC exploit for the prototype pollution vulnerability in Fig. 1, the main challenge is to construct a payload that, when passed to the vulnerable function, triggers a security relevant action. The goal of prototype pollution is to modify the prototype of objects. Exploits usually target the built-in Object prototype, since all objects inherit from it by default. To this end, PoCGen tests whether a property named "exploited" was added to the global `Object.prototype` object. In its first attempt, PoCGen generates an exploit that passes the object `{"__proto__": {"exploited": true}}` to the vulnerable function. However, this payload is invalid as it does not create the property `__proto__` on the passed object[1]. Exploitation of the vulnerability requires constructing an object that contains a property named `__proto__`. Once PoCGen executes the initial candidate exploit, it realizes that the exploit does not work, and by reasoning about the vulnerability, the code, and runtime information, the LLM comes up with a workaround. After multiple refinements, PoCGen generates a new exploit that uses `JSON.parse('{"__proto__": {"exploited": true}}')` to construct the payload that pollutes the `Object.prototype` object, as shown in Fig. 2.

Fig. 2: PoCGen-generated PoC exploit for CVE-2024-57063.

```javascript
async function exploit() {
  const DateFormatter=require("php-date-formatter");
  const maliciousJson='{"__proto__": {"exploited": true}}';
  const maliciousOptions=JSON.parse(maliciousJson);
  const result=new DateFormatter(maliciousOptions);
}
await exploit();
```

We run PoCGen on SecBench.js, a benchmark of currently 600 vulnerable npm packages with path traversal, prototype pollution, command injection, code injection, and ReDoS vulnerabilities. Since the SecBench.js benchmark only contains vulnerabilities up to 2022, we also evaluate PoCGen against a new dataset, that we extract from GitHub Advisory Database[2] and Snyk Vulnerability Database[3], which includes more recent vulnerabilities. Our results show that PoCGen successfully generates exploits for 432 out of 560 vulnerabilities in the SecBench.js dataset and 312 out of 794 in our new dataset. Our approach outperforms the state of the art by 45% on SecBench.js, while incurring the cost of only $0.02 per vulnerability.

By using PoCGen, developers of npm packages can generate PoC exploits for vulnerability reports they receive to help them understand the vulnerability and how to address it. They can also use the PoC exploits to test their patches and even add them to their test suites as regression tests. Moreover, security researchers can automate reporting vulnerabilities to downstream packages, by automating the PoC exploit generation process.

In summary, this paper makes the following contributions:

- A novel approach to autonomously generate and validate PoC exploits for vulnerabilities in npm packages using LLMs.
- Empirical evidence of the effectiveness of PoCGen on the SecBench.js dataset and more recent vulnerabilities.
- A new dataset of 794 vulnerabilities in npm packages, which includes more recent vulnerabilities and 312 PoC exploits generated by PoCGen.
- Insights into the influence of vulnerability type on the success of exploit generation.

II. Approach

Figure 3 provides an overview of PoCGen. In summary, PoCGen consists of four main components: (i) vulnerability information extraction, (ii) exploit generation, (iii) validation,

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#prototype_setter

[2] https://github.com/advisories

[3] https://security.snyk.io/

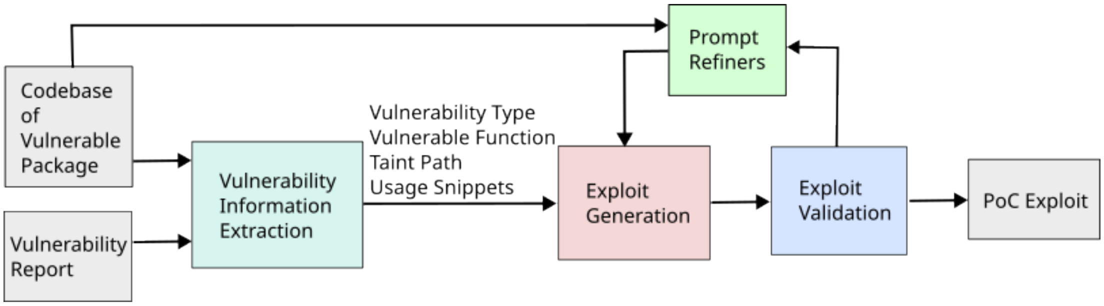


Fig. 3: Overview of PoCGen.

(iv) and prompt refinement, which are described in detail in the following sections.

PoCGen takes as input a vulnerability report, which is an informal description of the vulnerability. Such reports originate from several sources, such as vulnerability databases (e.g., the CVE database, GitHub Security Advisories, and Snyk), bug and issue trackers, or security mailing lists. The natural language description of the vulnerability in these reports is often vague and does not provide enough information to directly generate a PoC exploit. Therefore, in the first phase of our approach, PoCGen extracts some additional information from the vulnerability report and the codebase.

PoCGen extracts the vulnerability type, the likely vulnerable function, taint paths to vulnerable sinks, and usage examples. It then compiles them into a prompt for the LLM to generate a PoC exploit.

Once the LLM generates a candidate exploit, PoCGen executes and validates the candidate exploit using a set of runtime checkers, specific to each vulnerability type. For example, for command injection vulnerabilities, the validation checks whether a specific command is executed after running the candidate exploit, while for prototype pollution vulnerabilities, it checks whether a specific property is added to the global object.

Since the exploits are generated by LLMs, it is possible that the exploit does not use the vulnerable function to trigger the vulnerability, and calls some built-in functions that result in the same outcome. For example, in case of command injection, the call to `exec(<some command>)` can also result in the same outcome, but it does not invoke the vulnerable function. PoCGen checks the execution stack to detect such cases, and discards them as invalid exploits. Finally, to remove any invalid exploit that passed the validation, PoCGen checks whether the exploit actually triggers the vulnerability, by prompting the LLM with the vulnerability report and the generated PoC exploit.

If the candidate exploit is not valid, PoCGen refines the prompt using a set of refiners that provide static or dynamic information to the LLM to try to generate a valid exploit again. This process continues until either a valid exploit is generated or the budget is exhausted. If all validation checks pass and the LLM determines that the exploit correctly triggers the vulnerability, our approach considers it a valid PoC exploit.

A. Vulnerability Information Extraction

In this component, PoCGen extracts four pieces of information from the vulnerability report and the codebase to provide as context to the exploit generation component.

1) Vulnerability Type: First, PoCGen identifies the type of vulnerability. This is crucial to our approach as the type of vulnerability determines the goal of the exploit and how it should be validated. To do this, we prompt the LLM with the vulnerability report and ask it to identify the type of vulnerability. As the vulnerability report is written in natural language, and the description is informal, an LLM is a suitable tool to extract information from it. We provide the LLM with a list of possible vulnerability types, which are the five vulnerability types that we support in our approach, namely, path traversal, prototype pollution, command injection, code injection, and Regular Expression Denial of Service (ReDoS). PoCGen prompts the LLM to select the most relevant vulnerability type from the list.

2) Vulnerable Function: In our approach and throughout this paper, we refer to the function that is accessible to an attacker and is the entry point for the exploit as the *vulnerable function*. Finding the vulnerable function is helpful for generating the PoC exploit, as it provides information about the input types and possible values, and potentially the vulnerable execution path of the function. To identify the vulnerable function, we first load the package and dynamically extract all the functions that it exports. We then prompt the LLM with the vulnerability report and ask it to identify the vulnerable function from the extracted functions. All prompts and LLM responses are available in our supplementary material. Since the vulnerability report can be vague, and multiple functions can be candidates for the vulnerable function, we prompt the LLM to rank the functions based on their likelihood of being the vulnerable function.

3) Taint Path Extraction: To generate a successful PoC exploit, it is crucial to understand how the input to the vulnerable function flows through the code and reaches the sensitive operations, commonly referred to as vulnerable sinks. Taint analysis is a technique that tracks the flow of data through the code by marking a certain input as tainted and propagating this taint through the code. We use the static taint analysis provided by CodeQL[4].

In our approach we define a taint path as a sequence of source code lines starting with the signature line in the definition of the vulnerable function, and ending with a vulnerable sink. Any lines of code that propagate the taint are also included in the taint path. To provide more context to the LLM, for each line in the taint path, we include three lines before and after it as additional context. If these context windows overlap, we merge them to avoid duplication.

For each vulnerability type, we use the vulnerable sinks and taint propagation rules specified in the JavaScript security library of CodeQL[5].

From the vulnerable functions obtained in the previous step, in the order of their ranking, in batches of 50 functions, PoCGen queries the static taint analysis to extract at least one taint path, which is also used to pinpoint the vulnerable function. The taint analysis of CodeQL is designed for industry-level security analysis, which requires high precision and few false positives. This means that the taint analysis may not find all taint paths.

Hence, if the first taint tracking attempt does not find any taint paths, our approach retries the taint analysis with our own extended set of taint propagation rules and vulnerable sinks. If the extended taint analysis is also unsuccessful, PoCGen switches to a combination of static and dynamic taint analysis. First, it prompts the LLM to generate an exploit for the vulnerable function. Then, it executes the generated exploit and runs the static taint analysis on the functions that are executed during the exploit. If any of the executed functions have a taint path to a vulnerable sink, the approach uses this taint path as the taint path for the vulnerable function.

If PoCGen still do not find a taint path, it proceeds to the next batch of candidate functions from the ranking and repeats the process. If the approach exhausts all candidate functions without finding any taint paths, it considers the exploit generation attempt as failed and does not move to further steps of the approach.

The output of this phase is a sequence of code snippets interleaved with natural language explanations of how the code propagates the taint, shown in Fig. 4, with multiple sections if the taint path spans multiple files. In each section, there is a header specifying the file, followed by the taint path and its additional context from that file. The taint path lines are also marked by a comment at the end of the line.

4) Usage Snippets: To help the LLM generate a valid exploit, PoCGen extracts usage examples of the vulnerable function from the codebase. These examples allow the LLM to understand the function signature, and any setup that is required to call the function. We extract usage snippets both from the source code and from the documentation of the package. The usage snippets from the source code are extracted from test files using static analysis, by finding all the call sites of the vulnerable function. For the usage snippets from the documentation, we first extract all code pieces in the documentation, wrapped in triple backticks (```), and then we prompt an LLM to determine if they are usage examples of the vulnerable function. If they are, we also prompt the LLM to summarize them.

Fig. 4: Example of a taint path extracted by PoCGen.

~~~
Vulnerable method `import` of class `Environment` located in djv/lib/djv.js:
```js
import(config) { // tainted: "config"
  const item=JSON.parse(config) // tainted: "config"
  let restoreData=item // tainted: "item"
  if (item.name && item.fn && item.schema) {
    restoreData={
      [item.name]: item,
    }
  }
  Object.keys(restoreData).forEach((key)=>{ // tainted: "restoreData"
    const {name, schema, fn: source}=restoreData[key] // tainted: "restoreData"
    const fn=restore(source, schema, this.options) // tainted: "source"
    this.resolved[name]={
      name,
      schema,
```
Call to `restore`:
```js
function restore(source, schema, {inner}={}) { // tainted: "source"
  const tpl=new Function("schema", source)(schema) // tainted: "source"
  if (!inner) {
```
~~~

B. Exploit Generation

PoCGen assembles a prompt for the LLM to generate the PoC exploit. Figure 5 shows the prompt template that we use for generating the exploit. The prompt starts with naming the vulnerable function (as `vulnerableFunction`) and the vulnerability type (as `vulnerabilityType`), which are extracted in the previous phase, followed by a description of the vulnerability (as `vulnerabilityDescription`), which is the vulnerability report text. Then, we provide some example usages of the vulnerable function (as `usageSnippets`), which are extracted from the codebase. In the last section of the prompt, we describe how the generated exploit should look like and what it should do, by providing the skeleton of the exploit code (as `exploitSkeleton`), and also providing exploits of similar vulnerabilities (as `similarExploits`). The skeleton of the exploit code, is a fixed template that contains the definition of the exploit function, its call, and loading the vulnerable function from the vulnerable package. To extract similar exploits, our approach uses BM25 to find the three most similar vulnerability descriptions from the SecBench.js dataset to the current vulnerability report, and provides their exploits as examples.

[4] https://codeql.github.com/

[5] The module `semmle.javascript.security`

Fig. 5: Prompt template for the first round of exploit generation.

~~~
`{{vulnerableFunction}}` is vulnerable to {{vulnerabilityType}}.
## Vulnerability Description:
```
{{vulnerabilityDescription}}
```
{{> usageSnippets }}
{{> exploitSkeleton }}
## Task:
Create an exploit that uses the {{vulnerabilityType}} in `{{vulnerableFunction}}` to {{vulnerabilityTrigger}}.
Respond with the full exploit code and explain why it works.
If there is an exception thrown, do not try to handle it and pass it on.
Enclose the exploit code in backticks and define the exploit within a function named `exploit`.
{{> similarExploits }}
## Source code:
{{taintPathSnippets}}
~~~

C. Exploit Validation

We define the goal of the exploit specific to each vulnerability type. This allows us to validate the generated exploit with simple checkers. These checkers are inspired by the validation checks in SecBench.js [5].

a) Path Traversal.: The goal of path traversal exploits is to access a file in the root directory. In the validators of SecBench.js this was implemented as being able to read a flag file in the root directory. However, this would render exploits for vulnerabilities that perform other file operations invalid. To mitigate this, we consider all file system operations during runtime, and check the normalized path of the accessed file. If the normalized path is evaluated to `flag.txt` in a directory above the current directory, the exploit is considered valid.

b) Prototype Pollution.: In the case of prototype pollution vulnerabilities, the goal of the exploit is to add a property named `exploited` to the global `Object.prototype` object. However, this does not always guarantee that the exploit is valid, as the property can be added to the object without exploiting the vulnerability. To filter out such cases, we also check if the expression of the form `*[__proto__|prototype].exploited = *` is present in the exploit code. If such pattern exists, the exploit is considered invalid.

c) Command Injection.: For command injection vulnerabilities, the goal of the exploit is to execute the command `/usr/bin/genpoc`. This command writes a file to the file system, which is then checked by the validation checker.

d) Code Injection.: For code injection vulnerabilities, the goal of the exploit is to call the built-in `process.seteuid` function, with parameter 42. This indicates that the exploit was able to access the `process` module. Access to this module implies the ability to execute arbitrary code in the context of the Node.js process. To ensure that the exploit uses the vulnerable function, we statically check whether there is a direct call to `process.seteuid(42)` in the exploit code. If there is, we consider the exploit as invalid, as it does not exploit the vulnerable function.

e) ReDoS.: For ReDoS vulnerabilities, the goal of the exploit is to cause a denial of service by taking a long time to execute. We hook the string and regex functions in the V8 engine to measure the time these functions take. If the execution time of a function exceeds 1,500 milliseconds, we consider the exploit as valid.

As a last step in the validation process of any vulnerability type, we prompt the LLM to check whether the exploit actually triggers the vulnerability described in the report. This is done to filter out any invalid exploits that passed the previous validation checks.

D. Prompt Refinement

After the validation step, if the exploit is not valid and the maximum number of refinements is not exceeded, PoCGen refines the prompt to generate a new candidate exploit. PoCGen uses a set of refiners that provide static or dynamic information to the LLM to help it generate a valid exploit.

a) Context Refiners.: The first set of refiners are the *context refiners*, which provide additional context to the LLM to help it generate a valid exploit. Since the taint path only contains the taint propagation lines, checks on taint values are not included in the taint path. Therefore, the LLM might not have the information about the checks that are in place to prevent the vulnerability from being exploited. To address this, we provide a *body refiner*, which provides the full body of any function that has at least one line of code in the taint path.

However, there can be checks that happen via function calls that are not in the taint path. To address this, we also provide a *missing declaration refiner*, which provides the LLM with the ability to ask for definitions of variables and functions in the taint path, through the function calling format of OpenAI's API. The LLM can output a list of identifiers that it needs their definitions, and PoCGen will provide the definitions of these identifiers in the prompt.

b) Runtime Refiners.: The second set of refiners are *runtime refiners*, which add information about the execution to the prompt. The refiners in this category are the *error refiner*, the *coverage refiner*, the *debugger refiner*, and the vulnerability-specific refiners.

Since the exploit generated by the LLM can have runtime errors, for example from a wrong API usage, the *error refiner* provides the LLM with the error message that was thrown during the execution of the candidate exploit.

The *coverage refiner* provides the LLM with the coverage information of the candidate exploit, as markings in comments at the end of each line in the taint path. This information is useful for the LLM to understand which parts of the code were not executed. If the vulnerable sink was not executed, the information provided by this refiner can help the LLM to generate a new exploit that reaches the vulnerable sink.

The *debugger refiner* provides the LLM with a debugger-like tool. The LLM can output a list of expressions, for which it needs the runtime values. The refiner will provide the values of these expressions during the execution of the exploit in the prompt. These values are provided as comments in their respective lines in the taint path.

There are cases where the LLM generates an exploit that reaches the vulnerable sink, but the exploit fails the validation checks due to a wrong input. For path traversal, command injection, and code injection vulnerabilities, we provide specific refiners that hook into the vulnerable sinks and provide the runtime values passed to these functions. This form of feedback allows the LLM to understand how the input it generated is transformed, which can help it generate a valid exploit in the next iteration. For path traversal vulnerabilities, the refiner provides the values passed to the file system functions, like `fs.readFile` and `fs.open`. For command injection vulnerabilities, it provides the values passed to the `spawn` function. Finally, in case of code injection vulnerabilities, the refiner provides the values passed to the most common sink functions, like the `Function` constructor.

In every refinement attempt, PoCGen chooses one refiner from the front of a priority queue. Initially all refiners are in the queue. Each time the approach uses a refiner, it assigns a score based on the number of new errors the respective exploit causes, and the number of steps from the taint path it covers. It then adds the refiner to the priority queue with the score. Moreover, if a refinement generates a prompt that is already used, PoCGen does not query the LLM again, and moves to the next refiner.

To keep the prompts concise, in each refinement, PoCGen removes parts of the prompt that the LLM has correctly used in the previous attempts. For example, if the exploit generated in the previous step uses the vulnerable function correctly, PoCGen removes the usage snippets from the prompt.

III. Evaluation

We evaluate PoCGen on two datasets of vulnerabilities in npm packages to answer the following research questions:

- RQ1 How effective is PoCGen in generating PoC exploits for vulnerabilities in npm packages?
- RQ2 How much does each component of PoCGen contribute to the overall effectiveness?
- RQ3 How much does it cost to generate PoC exploits in terms of money and time?
- RQ4 What are the characteristics of vulnerabilities that affect the success of PoC generation?
- RQ5 How effective is PoCGen in generating PoC exploits for more recent and more diverse set of vulnerability reports?

A. Datasets

We use two datasets to evaluate PoCGen: the SecBench.js dataset [5], used in RQ1 to RQ4, and a new dataset, used in RQ5, which we extract from GitHub Advisory Database and Snyk Vulnerability Database. We refer to this new dataset as CWEBench.js.

TABLE I: Distribution of vulnerability types in CWEBench.js

Vulnerability Class	GHSA	Snyk	Total
Path Traversal	117	9	126
Prototype Pollution	221	27	248
Command Injection	178	2	180
Code Injection	83	1	84
ReDoS	156	0	156
Total	755	39	794

1) SecBench.js: The SecBench.js dataset contains 600 vulnerable npm packages with code injection, command injection, prototype pollution, path traversal, and ReDoS vulnerabilities. We exclude packages that have been removed from the npm registry. This leaves us with a total of 560 vulnerabilities to evaluate our approach.

2) CWEBench.js: The SecBench.js dataset only contains vulnerabilities up to 2022. Moreover, only vulnerabilities that (i) are exploitable on Linux, (ii) can be triggered by providing a single input, and (iii) their exploit creation required at most one hour, were included in the dataset. This means that the dataset has some inherent bias. Therefore, we create a new dataset that contains all vulnerabilities from 2013 to April 2025, that are not already in the SecBench.js dataset but match one of the five vulnerability types. This allows us to evaluate PoCGen on a more diverse set of vulnerabilities, with some of them more recent than vulnerabilities in SecBench.js, and some even newer than the training data of the LLM that we use.

To create the new dataset, *CWEBench.js*, we extract all vulnerabilities from the GitHub Advisory Database and Snyk Vulnerability Database that match one of the five vulnerability types by their CWE number. We use 22 and 35 for path traversal; 1321 for prototype pollution; 77 and 78 for command injection; 94 to 99 for code injection; and 400, 730, and 1333 for ReDoS.

As some vulnerabilities are mislabeled in the databases, we also extract all vulnerabilities that were not labeled with one of the CWE codes above with any of the following patterns:

- Path traversal: `travers[e|al]`
- Prototype pollution: `prototype`, `pollut[e|ion]`
- Command injection: `exec`, `execSync`, `shell injection`, `os injection`
- Code injection: `eval`, `code injection`, `code execution`
- ReDoS: `inefficient`, `regular expression`

We manually check these vulnerabilities and remove the ones that do not match one of the five vulnerability types. The remaining ones are added to the dataset.

Finally, we deduplicate the vulnerabilities in the dataset by their CVE IDs. The final dataset contains 794 vulnerabilities, as shown in Table I, where 794 of them are from GitHub Security Advisory database, and 40 from Snyk database.

B. Experimental Setup

We run all experiments on an Ubuntu 22.04 machine with Intel Zeon(R) Silver 4214 CPU, with 256 GB of RAM. The experimental setup uses Node.js version 22.11.0, running on a modified V8 engine that throws an error if a configurable backtracking limit is exceeded. For static taint analysis, we use CodeQL version 2.20.4.

The LLM that we use is OpenAI's `gpt-4o-mini-2024-07-18` model through the OpenAI API. We use a system prompt that assigns the role of a security researcher specialized in creating exploits for the identified security class to the LLM. This is done to reduce the refusals to generate exploits by the LLM for safety reasons. For each vulnerability, we allocate a time budget of one hour, a token budget of 300k input tokens, and 100k output tokens. The maximum refinement budget is set to 30 iterations.

We compare PoCGen against two baselines: Explode.js [12] and an LLM-based agent using the AutoGPT framework[6]. Explode.js is a state-of-the-art approach for generating PoC exploits. It first uses a static dataflow analysis to detect which exported functions reach a vulnerable sink, which is then used to create an exploit template. Then, using symbolic execution, it generates symbolic inputs to exploit the vulnerability. Finally, it uses an SMT solver to generate concrete inputs that trigger the vulnerability.

Since PoCGen is the first LLM-based PoC exploit generator, we implement an LLM-based agent using the AutoGPT framework as a second baseline. Recently, LLM-based agents have shown great promise in software engineering tasks, such as resolving issues [13], repairing bugs [14], and executing arbitrary projects [15]. The AutoGPT agent can use tools to traverse the codebase, such as navigating the file system, reading and writing files, and executing shell commands, by default. We also add two tools to allow direct execution of a JavaScript code piece or a JavaScript file for this agent.

C. RQ1: Effectiveness

We evaluate the effectiveness of PoCGen in generating PoC exploits for vulnerabilities in SecBench.js as done in previous work [12]. We measure the success rate of our approach and compare it to the baselines. We also report the number of failed attempts, and the number of false positives, which are exploits that pass the validator, but do not trigger the vulnerability through the vulnerable function. For each PoC exploit that PoCGen generates, one of the authors manually inspects it to determine whether it is a false positive or a successful exploit.

In our experiments, we run PoCGen on all 560 vulnerabilities in SecBench.js, and use the reported results of Explode.js[7] on the same set of vulnerabilities. However, since agentic approaches are slower and more expensive, we limit the number of vulnerabilities evaluated with our agentic baseline approach to 100. We randomly sample 20 vulnerabilities from each of the five vulnerability types in SecBench.js, and run AutoGPT on them.

Figure 6 shows the effectiveness of PoCGen, Explode.js, and AutoGPT on the SecBench.js dataset. PoCGen successfully generates PoC exploits for 432 out of 560 vulnerabilities, which is 77% of the vulnerabilities. Explode.js successfully generates PoC exploits for 182 out of 560 vulnerabilities, which is 32% of the vulnerabilities. AutoGPT generates successful PoC exploits for 16 out of the 100 vulnerabilities. For the same set of 100 vulnerabilities, PoCGen generates PoC exploits for 63 vulnerabilities. The results show that PoCGen outperforms Explode.js by 45 and AutoGPT by 47 absolute percentage points.

When comparing the performance of PoCGen on different vulnerability types, we find that PoCGen performs best on path traversal, prototype pollution, and command injections, with success rates of above 83%. Explode.js also performs best on these vulnerability types, with success rates between 50% and 60%. Since Explode.js does not support ReDoS vulnerabilities, its success rate on ReDoS vulnerabilities is 0%. AutoGPT performs best on prototype pollution and command injection, with success rates of 35% and 25%, respectively. AutoGPT performs worst on path traversal, with no successful PoC exploits.

A closer comparison of the sets of vulnerabilities covered by each approach reveals that Explode.js generates PoC exploits for only two command injection vulnerabilities that are not handled by PoCGen. For one of these cases, the vulnerability can be exploited with a simple payload. But, our validator requires the execution of the command `/usr/bin/genpoc`, and our prompts direct the LLM to generate an input that executes this command. Therefore, the exploit becomes much more complex, and the LLM is not able to generate it. The other case is a vulnerable function that implements the functionality to kill processes. Our generated payload causes the running bash process to be killed, which does not allow the execution of the rest of the command. The LLM is also not able to fix this after the refinements.

On the other hand, PoCGen generates PoC exploits for 158 vulnerabilities that Explode.js does not generate PoC exploits for. PoCGen generates PoC exploit for all vulnerabilities that AutoGPT generates PoC exploits for.

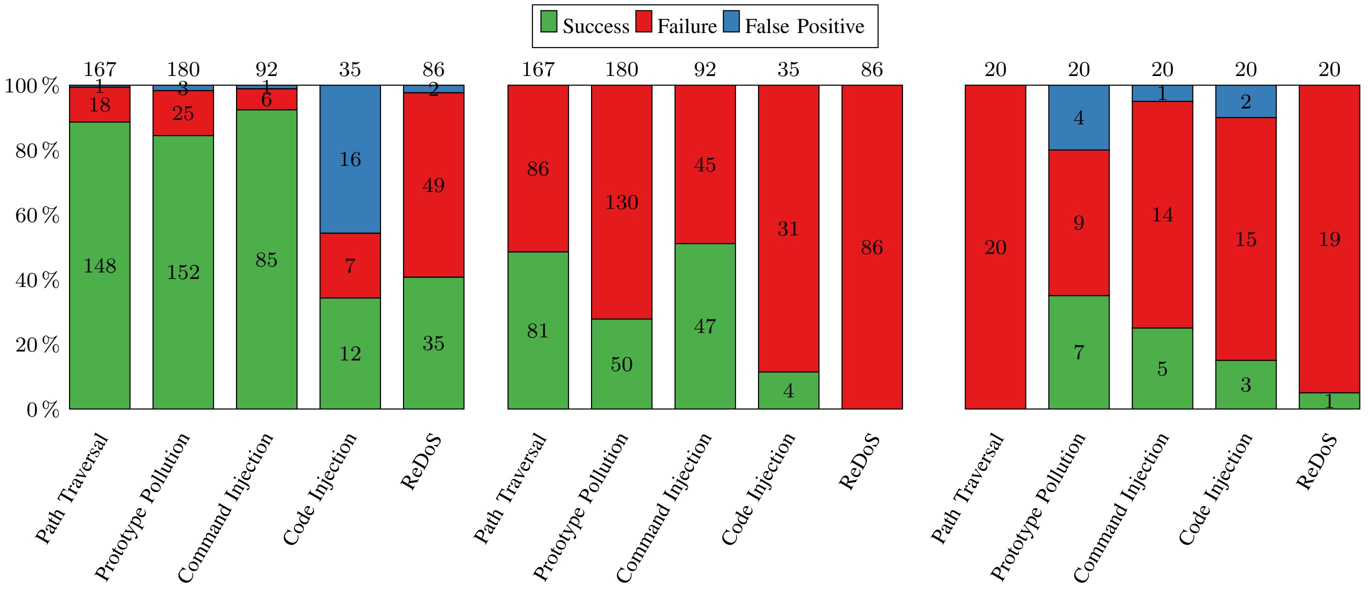
PoCGen uses multiple approaches to reduce the number of false positives, i.e., exploits that pass the validation and LLM checks, but do not correctly trigger the vulnerability. As a result, it only generates 23 false positive PoC exploits, which amounts to just 5% of the successful exploits generated.

D. RQ2: Ablation Study

To evaluate the impact of each component on the effectiveness of PoCGen, we perform an ablation study on the SecBench.js dataset. We evaluate the following configurations

[6] We use the open-source version of AutoGPT, now called AutoGPT Classic, which is available at https://github.com/Significant-Gravitas/AutoGPT/tree/793d056d81ca1c1a21538ddeef13c4e6d7d0d254/classic

[7] https://github.com/formalsec/explode-js/blob/71ec17fe90f29e236b01b8cad02685344f8aff10/bench/explode-vulcan-secbench-results.csv



(a) PoCGen (b) Explode.js (c) AutoGPT

Fig. 6: Effectiveness of PoCGen, Explode.js, and AutoGPT on SecBench.js.

TABLE II: The effect of information provided in the prompt of PoCGen on successfully generating PoC exploits.

Configuration	Valid Exploits	Success Rate
PoCGen	**432**	**77%**
noTaintPath	421	75%
noUsageSnippet	196	39%
noFewShot	402	72%

of PoCGen to measure the impact of different information about the vulnerability provided in the prompt.

- PoCGen: The complete PoCGen approach with all components.
- noTaintPath: PoCGen without the taint path extraction, described in Section II-A3. In this configuration, the LLM prompt does not contain the taint path, and also the context refiners that depend on the taint path are not used.
- noFewShot: PoCGen without the few-shot examples of similar vulnerabilities' exploits, described in Section II-B.
- noUsageSnippet: PoCGen without the usage snippet examples, described in Section II-A4.

We also evaluate the impact of the refiners on the effectiveness of PoCGen.

The success rates of these configurations on SecBench.js are shown in Table II. These results show that all the information provided in the prompt contribute to PoCGen's overall performance, with the usage snippets having the highest impact, followed by the few-shot examples.

Moreover, the prompt refiners also have a significant impact on the performance of PoCGen. Only 36% of the successful PoC exploits are generated in the first attempt, and 51% are generated with two to 10 refinements. On average, successful PoC exploits are generated in 3.92 rounds of refinement.

E. RQ3: Costs

To evaluate the costs of PoCGen, we measure the time and token usage of the approach. The time and token usage measurements are done through the experiments on the SecBench.js dataset. On average each PoC exploit generation attempt takes 11 minutes. Since successful attempts stop earlier than failing attempts, which require running the refiners, the successful runs complete on average in 7 minutes. A significant portion of this time is spent on the LLM API calls, which take 41% of the execution time. On average the taint path analysis with the initial CodeQL setup takes 21%, and the API exploration takes 3% of the execution time.

On average, for each attempt to generate an exploit, PoCGen sends 61,234 tokens to the LLM API and receives 17,750 tokens in response. With the current OpenAI API pricing as of May 2025, this costs $0.02 per exploit generation attempt. For the successful attempts, this cost is $0.008.

F. RQ4: Qualitative Analysis

In this section, we present examples of successful and unsuccessful attempts by PoCGen to generate PoC exploits.

The vulnerability report `CVE-2023-26127` describes a command injection vulnerability in the `n158` package. This vulnerability is due to not sanitizing the input, which then flows to the `exec` function of `shelljs`. However, the input is first passed ot a copy function, as show in Fig. 7. This means that if the `seedPath` is set to a value where the corresponding path does not exist, the exploit fails before reaching the vulnerable sink. This constraint is not easy to capture with symbolic execution as attempted by Explode.js,

Fig. 7: Example of a command injection vulnerability in the `n158` package.

```
module.exports = function (seedPath, targetPath) {
  fsExtra.copySync(seedPath, targetPath);
  shell.exec('sh -c \'cd '+targetPath+' && npm install\'');
  shell.exec('sh -c \'cd '+targetPath+' && npm start\'');
}
```

Fig. 8: Example of a successful PoC exploit generation by PoCGen.

```
async function exploit() {
  const initProject = require("n158/src/cli/initProject");
  const seedPath = "/path/to/valid/seed";
  const targetPath = "targetPath; /usr/bin/genpoc";
  const fs = require('fs');
  if (!fs.existsSync(seedPath)) {
      fs.mkdirSync(seedPath, { recursive: true });
  }
  const result = await initProject(seedPath, targetPath);
}
await exploit();
```

but using execution feedback and an LLM, PoCGen is able to generate a working exploit.

In the first round of exploit generation, PoCGen generates an exploit that calls `initProject("someSeedPath", "targetPath; /usr/bin/genpoc")`. After the execution of this exploit, the runtime feedback shows that there was a runtime error happening at the `fsExtra.copySync` function, which is due to the fact that no directory called `"someSeedPath"` exists. The LLM then naively tries to fix this by changing `"someSeedPath"` to `"/path/to/valid/seed"`, which results in the same error. It takes multiple refinements until the LLM requests the definition of `shelljs`, which allows the reasoning part of the response to understand that it needs to create the directory for a successful exploit. This results in the final exploit shown in Fig. 8.

The vulnerability report `GHSA-3486-rvxc-hrrj` describes a command injection vulnerability in the `gitblame` package. The package exports one function, which takes a file path as an argument and after some processing passes it to `exec`. The relevant parts of the source code are shown in Fig. 9. PoCGen is not able to generate a working exploit for this vulnerability. The reason is that our validator requires the execution of the command `/usr/bin/genpoc`. However, the split by `path.basename` and `path.dirname` does not allow passing of payloads with ”/” to reach the sink without change. This means that our approach is limited by the constraints imposed by the validator. Even with multiple refinements, PoCGen is not able to generate a working exploit, although the exploit for this vulnerability in SecBench.js has the simple payload ”`& touch gitblame`”.

Fig. 9: Example of a command injection vulnerability in the `gitblame` package.

```
module.exports = function (file, cb) {
  var dirname = path.dirname(file);
  var filename = path.basename(file);
  exec('git blame ' + filename, {cwd: dirname}, ...
```

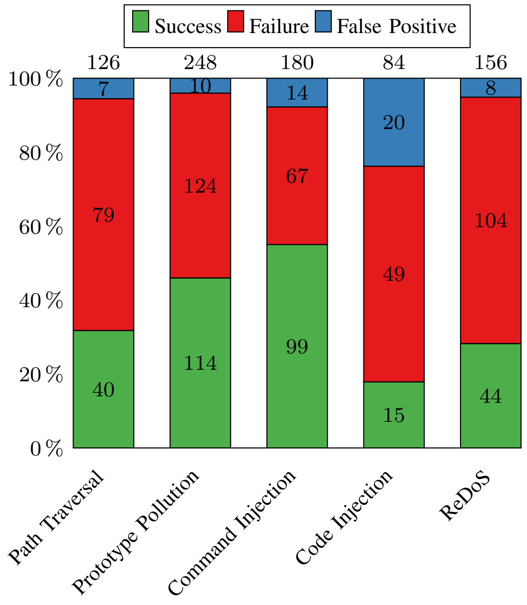


Fig. 10: Effectiveness of PoCGen on CWEBench.js.

G. RQ5: Generalizability

To evaluate the generalizability of our approach to new vulnerabilities, we run PoCGen on the CWEBench.js dataset. We use the same setup as in Section III-C and run PoCGen on all 794 vulnerabilities in the dataset.

Figure 10 shows the effectiveness of PoCGen on the CWEBench.js dataset. PoCGen generates PoC exploits for 312 out of 794 vulnerabilities, which is 37% of the vulnerabilities.

The difference in the results on SecBench.js and CWEBench.js comes from two features of these datasets. First, the criteria to include a vulnerability in the SecBench.js dataset was that the authors of SecBench.js could generate a PoC exploit in one hour, which means some complicated exploits were excluded from the dataset. Second, the vulnerabilities in the SecBench.js dataset were required to work on a specific environment, and that an input should trigger a security-relevant action. This means that some vulnerabilities in CWEBench.js are not compatible with our execution environment, as it is very similar to the environment used in SecBench.js. For example, our approach failed for 39 out of 794 vulnerabilities in CWEBench.js due to errors encountered during the installation of the packages. Furthermore, some vulnerabilities require triggers other than calling a function or sending a request to an API endpoint, such as setting a malicious SSID for a Wi-Fi network as in CVE-2023-42810. Yet, despite these differences, PoCGen is able to generate PoC exploits for a significant number of vulnerabilities in CWEBench.js.

Moreover, we split the vulnerabilities in CWEBench.js into vulnerabilities before and after the knowledge cutoff of the gpt-4o-mini model. The results show that PoCGen successfully generates PoC exploits for 35% of the vulnerabilities before the knowledge cutoff, and 41% of the vulnerabilities after the knowledge cutoff. This shows that PoCGen generalizes well to newer vulnerabilities.

IV. Threats to Validity

The first threat to internal validity is the LLM's potential to recall exploits from its training data. To mitigate this, we use the same LLM for the AutoGPT baseline, but we also evaluate PoCGen on newer vulnerabilities that are not in the training data of the LLM. The results shows that the memorization effects are negligible. By using the CWEBench.js dataset, we also mitigate the external validity threat of dataset bias from SecBench.js. Finally, the labeling process of false positive PoC exploits is done manually, which can introduce human bias.

V. Related Work

a) Vulnerability Detection: Greybox fuzzing [16], [17], applied to source code [18] or binaries [19], is a common approach for vulnerability detection. To evaluate fuzzing, techniques for reverting fixes [20] and benchmarking methodologies [21], [22] have been proposed. Learning-based vulnerability detection includes neural classification [23]–[25], graph neural networks [26], [27], and combinations of LLMs with static analysis. Similar to our work, the latter leverages CodeQL to identify taint flows [28]. To support learning-based detection, datasets from commit histories [29] and large-scale vulnerability generation approaches [30], [31] have been introduced. Vulnerability detection is orthogonal to our work, as we assume vulnerabilities are already described in a report but lack an exploit.

b) Detecting and Exploiting Node.js Vulnerabilities: The closest work to PoCGen is Explode.js [12], which finds vulnerabilities and generates PoC exploits for npm packages. Explode.js uses static dataflow analysis to extract the sequence of function calls required to propagate attacker input to a vulnerable sink. It then applies symbolic execution and constraint solving to generate a PoC exploit. However, Explode.js does not model external functions and libraries during symbolic execution, which limits its effectiveness, as the npm ecosystem heavily relies on small, reusable packages. PoCGen addresses this limitation by leveraging LLMs to generate exploits and by incorporating runtime feedback. LLMs, trained on large code corpora, can better predict the behavior of external functions and reason about inputs that exercise specific program paths.

Other approaches have used symbolic execution to generate exploits for vulnerabilities. FAST [32] applies bi-directional dataflow analysis to detect taint paths efficiently, enabling scalable vulnerability detection. It generates exploits by concretizing symbolic path constraints once a vulnerability is found. Node-Medic [33] and Node-Medic-FINE [11] combine dynamic taint analysis with symbolic execution to detect and generate exploits for Node.js packages. Node-Medic-FINE further incorporates fuzzing to generate inputs and explore additional execution paths. All three approaches are outperformed by Explode.js [12], which we therefore use as a baseline in our evaluation.

c) Test Generation for Security Vulnerabilities: Zhang et al. [34] and Gao et al. [35] use LLMs to generate unit tests for Java vulnerabilities given a PoC exploit. Their goal is to encourage developers to update vulnerable dependencies and prevent supply chain attacks. In contrast, our work generates code that directly exploits a vulnerable package, rather than exploiting it through a third-party dependency.

d) LLM-Assisted Attacks: Recent work has explored the potential of LLMs for attacking vulnerable software. PentestGPT [36] uses LLMs for penetration testing. Xu et al. [37] developed an LLM agent with command-line access to exploit vulnerabilities in Linux and Windows applications. Charan et al. [38] investigated using LLMs to generate payloads for exploiting vulnerabilities.

e) Vulnerability Mitigation and Repair: Mitigating vulnerabilities can involve removing unused dependencies [39] or reducing the privileges of vulnerable code [40]. Repairing vulnerabilities can be achieved by fine-tuning LLMs to find fixes [41], using LLM agents [42], or applying generative adversarial networks (GANs) [43].

f) JavaScript and Npm Ecosystem Security: The npm ecosystem faces various security issues, such as injection attacks [44], ReDoS [45], and malicious packages [46]. Several empirical studies have analyzed npm from a security perspective, including vulnerability propagation [2], [47] and how developers address vulnerabilities [3]. Householder et al. [4] found that most vulnerability reports lack a public PoC exploit for at least one year. Yadmani et al. [6] further showed that many PoC exploits on GitHub are themselves malicious. These findings motivate our work on automated PoC exploit generation. To support further research, Bhuiyan et al. [5] introduced the SecBench.js dataset, and Brito et al. [48] created the VulcaN dataset.

VI. Conclusion

In this paper, we presented PoCGen, an LLM-based approach to automatically generate proof-of-concept exploits for vulnerabilities in npm packages. PoCGen extracts information from the vulnerability report and the codebase to generate a PoC exploit using an LLM. The generated PoC exploits are validated using a set of runtime checkers, and the prompt is refined using static and dynamic information to generate a valid exploit. We evaluated PoCGen on two datasets of vulnerabilities in npm packages, SecBench.js and a new dataset that we create. PoCGen generates PoC exploits for 77% of the vulnerabilities in SecBench.js, outperforming the state-of-the-art Explode.js by 35 and AutoGPT by 47 absolute percentage points. We also evaluated PoCGen on the new dataset, where it generates PoC exploits for 37% of the vulnerabilities.

By automating the generation of PoC exploits, PoCGen enables developers and security teams to more rapidly understand and address vulnerabilities, reducing the time between vulnerability disclosure and patch deployment. This automation also improves regression testing, as the generated PoC exploits can be directly used to verify the effectiveness of fixes and to prevent vulnerabilities from reappearing in future releases. For security researchers, PoCGen provides an automated way to evaluate the effectiveness of existing mitigation strategies across large sets of vulnerabilities. Furthermore, PoCGen can improve the quality of existing vulnerability reports, including those that are poorly documented or lack existing exploits. Finally, automated PoC generation can facilitate responsible vulnerability disclosure by providing clear, actionable evidence to affected parties, encouraging timely remediation.

Data Availability

The source code of PoCGen, the new dataset, and all experiment scripts are available at https://figshare.com/s/b1c0d41348c353fc2033.

References

[1] "CVE: Common Vulnerabilities and Exposures," https://www.cve.org/about/Metrics, accessed: 2025-05-14.

[2] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

[3] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are Weak Links in the npm Supply Chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, May 2022, pp. 331–340.

[4] A. D. Householder, J. Chrabaszcz, T. Novelly, and D. Warren, "Historical Analysis of Exploit Availability Timelines," 2020.

[5] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, May 2023, pp. 1059–1070.

[6] S. E. Yadmani, R. The, and O. Gadyatskaya, "Beyond the Surface: Investigating Malicious CVE Proof of Concept Exploits on GitHub," Jun. 2023.

[7] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, Jan. 2024.

[8] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *45th International Conference on Software Engineering, ser. ICSE*, 2023.

[9] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-End Program Repair with LLMs," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. San Francisco CA USA: ACM, Nov. 2023, pp. 1646–1656.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death." in *ACM Conference on Computer and Communications Security*, 2006, pp. 322–335.

[11] D. Cassel, N. Sabino, M.-C. Hsu, R. Martins, and L. Jia, "NodeMedic-FINE: Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities," in *Proceedings 2025 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2025.

[12] F. Marques, M. Ferreira, A. Nascimento, M. E. Coimbra, N. Santos, L. Jia, and J. F. Santos, "Automated exploit generation for node.js packages," in *PLDI*, 2025.

[13] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html

[14] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," Preprint, 2024.

[15] I. Bouzenia and M. Pradel, "You name it, I run it: An LLM agent to execute tests of arbitrary projects," in *ISSTA*, 2025.

[16] M. Zalewski, "American fuzzy lop (afl)," https://lcamtuf.coredump.cx/afl/, 2013. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[17] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Trans. Software Eng.*, vol. 45, no. 5, pp. 489–506, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2017.2785841

[18] G. Sherman and S. Nagy, "No harness, no problem: Oracle-guided harnessing for auto-generating c api fuzzing harnesses," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 775–775.

[19] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1497–1511. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00009

[20] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "Fixreverter: A realistic bug injection methodology for benchmarking fuzz testing," in *Proceedings of the 31st USENIX Security Symposium*, 2022.

[21] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[22] Y. Li, S. Ji, Y. Chen, S. Liang, W. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *USENIX Security*, 2021.

[23] Z. Li, S. X. Deqing Zou and, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *NDSS*, 2018.

[24] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR*. ACM, 2022, pp. 608–620. [Online]. Available: https://doi.org/10.1145/3524842.3528452

[25] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Software Eng.*, vol. 48, no. 9, pp. 3280–3296, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3087402

[26] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 292–303. [Online]. Available: https://doi.org/10.1145/3468264.3468597

[27] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[28] Z. Li, S. Dutta, and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities," 2024.

[29] 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

[30] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "Generating realistic vulnerabilities via neural code editing: An empirical study," in *ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022.

[31] Y. Nong, Y. Ou, M. Pradel, F. Chan, and H. Cai, "Vulgen: Realistic vulnerability generation via pattern mining and deep learning," in *ICSE*, 2023.

[32] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao, "Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 1059–1076.

[33] D. Cassel, W. T. Wong, and L. Jia, "NodeMedic: End-to-End Analysis of Node.js Vulnerabilities with Provenance Graphs," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, Jul. 2023, pp. 1101–1127.

[34] Y. Zhang, W. Song, Z. Ji, Danfeng, Yao, and N. Meng, "How well does LLM generate security tests?" Oct. 2023.

[35] Y. Gao, X. Hu, Z. Chen, and X. Yang, "Vulnerability-Triggering Test Case Generation from Third-Party Libraries," Feb. 2025.

[36] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "{PentestGPT}: Evaluating and Harnessing Large Language Models for Automated Penetration Testing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 847–864.

[37] J. Xu, J. W. Stokes, G. McDonald, X. Bai, D. Marshall, S. Wang, A. Swaminathan, and Z. Li, "AutoAttacker: A Large Language Model Guided System to Implement Automatic Cyber-attacks," Mar. 2024.

[38] P. V. S. Charan, H. Chunduri, P. M. Anand, and S. K. Shukla, "From Text to MITRE Techniques: Exploring the Malicious Use of Large Language Models for Generating Cyber Attack Payloads," May 2023.

[39] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 121–134. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/koishybayev

[40] N. Vasilakis, C. 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 *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 1821–1838. [Online]. Available: https://doi.org/10.1145/3460120.3484535

[41] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. T. Vechev, "Deepcode AI fix: Fixing security vulnerabilities with large language models," *CoRR*, vol. abs/2402.13291, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.13291

[42] Y. Zhang, J. Wang, D. Berzin, M. Mirchev, D. Liu, A. Arya, O. Chang, and A. Roychoudhury, "Fixing security vulnerabilities with ai in oss-fuzz," 2024. [Online]. Available: https://arxiv.org/abs/2411.03346

[43] J. Harer, O. Ozdemir, T. Lazovich, C. P. Reale, R. L. Russell, L. Y. Kim, and S. P. Chin, "Learning to repair software vulnerabilities with generative adversarial networks," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, 2018, pp. 7944–7954. [Online]. Available: http://papers.nips.cc/paper/8018-learning-to-repair-software-vulnerabilities-with-generative-adversarial-networks

[44] C.-A. Staicu, M. Pradel, and B. Livshits, "Understanding and automatically preventing injection attacks on Node.js," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[45] C. Staicu and M. Pradel, "Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *USENIX Security Symposium*, 2018, pp. 361–376.

[46] A. Sejfia and M. Schaefer, "Practical automated detection of malicious npm packages," in *ICSE*, 2022.

[47] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," in *ICSE*, 2022.

[48] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. F. Santos, and N. Santos, "Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages," *IEEE Transactions on Reliability*, vol. 72, no. 4, pp. 1324–1339, Dec. 2023.