

FAULTLINE: Automated Proof-of-Vulnerability Generation using LLM Agents

Vikram Nitin**
vikram.nitin@columbia.edu
Columbia University
New York, NY, USA

Baishakhi Ray
rayb@cs.columbia.edu
Columbia University
New York, NY, USA

Roshanak Zilouchian
Moghaddam
rozilouc@microsoft.com
Microsoft
Redmond, WA, USA

Abstract

Despite the critical threat posed by software security vulnerabilities, reports are often incomplete—lacking the proof-of-vulnerability (PoV) tests needed to validate fixes and prevent regressions. These tests are crucial not only for ensuring patches work, but also for helping developers understand exactly how vulnerabilities can be exploited. Generating PoV tests is a challenging problem, requiring reasoning about the flow of control and data through deeply nested levels of a program.

We present FAULTLINE, an LLM agent workflow that uses a set of carefully designed reasoning steps, inspired by aspects of traditional static and dynamic program analysis, to automatically generate PoV test cases. Given a software project with an accompanying vulnerability report, FAULTLINE 1) traces the flow of an input from an externally accessible API (“source”) to the “sink” corresponding to the vulnerability, 2) reasons about the conditions that an input must satisfy in order to traverse the branch conditions encountered along the flow, and 3) uses this reasoning to generate a PoV test case in a feedback-driven loop. FAULTLINE does not use language-specific static or dynamic analysis components, which enables it to be used across programming languages.

To evaluate FAULTLINE, we collate a challenging multi-lingual dataset of 100 known vulnerabilities in Java, C and C++ projects. On this dataset, FAULTLINE is able to generate PoV tests for 16 projects, compared to just 9 for CodeAct 2.1, a popular state-of-the-art open-source agentic framework. Thus, FAULTLINE represents a 77% relative improvement over the state of the art. Our findings suggest that hierarchical reasoning can enhance the performance of LLM agents on PoV test generation, but the problem in general remains challenging even for state-of-the-art models. We make our code and dataset publicly available in the hope that it will spur further research in this area.¹

*Work done when the author was an intern at Microsoft

¹<https://github.com/faultline-pov/icse-26>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXXX>

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Automated static analysis*; *Compilers*; *Software verification and validation*; • **Information systems** → *Language models*.

Keywords

Test generation, Vulnerability Detection, Agents, LLMs

ACM Reference Format:

Vikram Nitin, Baishakhi Ray, and Roshanak Zilouchian Moghaddam. 2018. FAULTLINE: Automated Proof-of-Vulnerability Generation using LLM Agents. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 14 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Security vulnerabilities pose a significant threat to the software development process, driving the community to build various automated detection and fixing tools [14]. When a vulnerability is detected in a project, it is reported to the developers of the project, who then attempt to quickly fix it. Subsequently, a report is generated in the National Vulnerability Database (NVD) [24], containing a textual description of the vulnerability and mitigation strategies for users of the software. However, most of these reports lack Proof-of-Vulnerability (PoV) tests that demonstrate the vulnerability. PoV tests are designed to *fail* when the vulnerability exists, and *succeed* when the vulnerability is fixed. Thus, they act as an oracle to verify the effectiveness of the fix, and ensure that the vulnerability is not inadvertently reintroduced during future development of the project. In addition, they can enable developers to better understand the vulnerability. Studies have shown [23] that human developers struggle to reproduce vulnerabilities from reports, because these reports frequently miss crucial information. PoV tests complement the information in a report and provide a clear demonstration of the exploit.

Existing work and limitations: Recently, Large Language Models (LLMs) have been used as components in *autonomous agents* to solve various software engineering tasks [31, 32, 34]. These systems augment LLMs with the ability to invoke *tools* to read, write and execute code, enabling the LLM to interact with the code base much as a human developer would. However, constructing PoV tests is a challenging problem for LLM agents [35]. Some of the reasons for this are:

- *Insufficient understanding of data flow.* A vulnerability exploit starts with an externally accessible API or user input (“source”), and traverses through multiple function calls until it reaches the

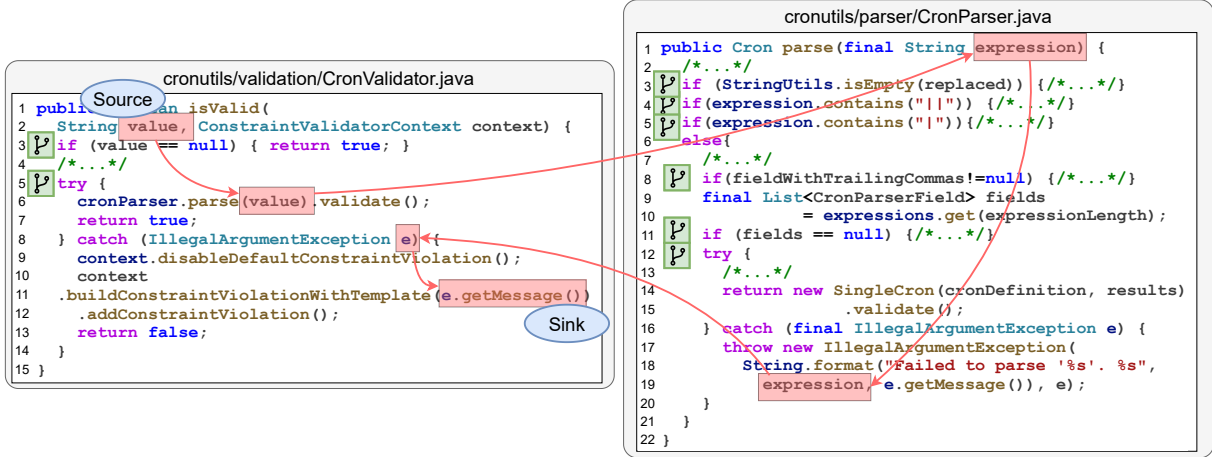


Figure 1: Motivating example - A Code Injection vulnerability from the `cron-utils` Java library. The vulnerable flow is highlighted in red, and the branch conditions are marked in green. The `String value` is taken as a user-provided argument on line 2 of `CronValidator.java`, and it flows to the Sink on line 11 where any embedded code could potentially be executed.

location where the vulnerability occurs (“sink”). Writing a PoV test involves reasoning about this flow, and invoking the precise methods that trigger it. LLMs are usually not trained on data flow traces, and therefore they do not effectively leverage this type of reasoning to solve programming tasks.

- *Insufficient understanding of control flow.* The program path from source to sink frequently involves many branch conditions, that divert the flow into paths that do not reach the vulnerability. The input in a PoV test must be carefully crafted such that the program flow proceeds along the correct path at each branch. LLM agents often miss certain crucial conditions on test inputs, and the tests do not reach the vulnerability. Additionally, they are unable to systematically reason about the cause of this failure and refine the test.

- *Misalignment with initial goals.* A PoV test must satisfy certain requirements - it must fail when the vulnerability exists, and demonstrate the exploit by actually running the vulnerable code. LLM agents frequently stop after generating a test that satisfies *some* (but not all) of these requirements. For instance, they may generate a test that simply reads the source code to check for the presence of a particular line of code corresponding to the vulnerability, and the test does not actually build the project or run the vulnerable code.

Our approach: To address the above shortcomings, we propose FAULTLINE, a workflow-based LLM agent that uses a composition of carefully designed reasoning steps to design a PoV test for a known vulnerability. Unlike existing agents that generate tests with an incomplete or incorrect understanding of program semantics, FAULTLINE prompts an LLM to extract certain semantic properties of the program before generating a test. Specifically, given a program along with an accompanying vulnerability report, FAULTLINE traces the flow of data from source to sink, reasons about the requirements that a test must satisfy in order to cover this path, uses these insights to generate a PoV test, and refines it in a feedback-guided loop.

Results summary: To evaluate FAULTLINE, we collate a challenging multi-lingual dataset of 100 known vulnerabilities in Java, C and C++ projects. Our key findings are listed below:

- On this dataset, FAULTLINE is able to generate correct PoV tests for 16 projects. In comparison, CodeAct 2.1, a popular agentic framework, is able to generate correct PoV tests for only 9.
- The tests generated by FAULTLINE reach the functions or methods corresponding to the vulnerability for 31 projects, compared to 21 for the baseline.
- We show that both flow reasoning and branch reasoning are essential to FAULTLINE’s performance.

Contributions: This paper makes the following contributions to the state of the art:

- (1) We design an agentic workflow based on a series of carefully crafted reasoning steps to generate PoV tests.
- (2) We empirically establish the effectiveness of this workflow in generating PoV tests, and highlight the importance of each component of the workflow.
- (3) We contribute a benchmark for PoV test generation, comprising 100 vulnerabilities spanning 4 CWE categories. This dataset challenges LLMs to reason about extremely subtle properties of a program, and represents a frontier for LLM-based code reasoning.

2 Background and Motivating Example

2.1 Background

A security vulnerability in a software project manifests as a *flow* that leads from a “source” to a “sink”. A **source** is traditionally defined as a program point where data enter the program from external or untrusted sources. Some examples are external API functions (for software libraries), user form inputs (for web applications), or HTTP endpoints (for web services). More generally, any property of the program that can be controlled by an attacker can be considered a source. A **sink** is any program construct that can cause undesirable effects if attacker-controlled data is passed directly to it. For instance, a function that executes SQL queries is considered a sink because it can be invoked with SQL queries that delete or alter data stored in the linked database.

The fundamental principle of secure software design is to ensure that each flow between a source and a sink is properly filtered, or **sanitized**. Consider a web form that accepts text input from a user, and uses this text to retrieve matching records from a database. A quintessential vulnerability pattern in such a setting is *SQL injection*, whereby an attacker embeds carefully crafted SQL queries in their input text. If this input text reaches a function that interacts with the database, the embedded SQL queries might be executed on the underlying database, leading to data loss or privacy concerns. To safeguard against this, developers must sanitize text inputs by checking for patterns that are indicative of SQL injection exploits.

Software vulnerabilities can be organized into groups depending on the nature of the sources and sinks. Common Weakness Enumeration (CWE) [22] is a widely adopted categorization system for vulnerabilities. Each category is assigned a number, e.g., CWE-94 corresponds to *Code Injection* vulnerabilities, such as SQL injection. When a vulnerability is reported, it is usually assigned one or more CWE categories.

2.2 Motivation

In this section, we use a real-world example to highlight the challenges involved in generating PoV test cases. Figure 1 shows a *Code Injection* vulnerability in the Java library `cron-utils`. An attacker can exploit this to execute arbitrary code on the host system. When one tries to generate a PoV for this vulnerability, certain challenges arise:

Challenge 1: *Tracing the flow of data from source to sink.* A prerequisite for generating a PoV test is identifying a source and sink with an un-sanitized flow between them. In Figure 1, the function `isValid` can be called with a user-provided String `value`, so this is a *source*. The function `buildConstraintViolationWithTemplate` on line 11 of `CronValidator.java` accepts arguments written in Java's Expression Language (EL), and can execute arbitrary code. So this is a *sink*. When we look at `CronValidator.java`, it may seem like there is no flow between `value` on line 2 and `e` on line 11. However, when we look closer, we find that `cronParser.parse()` on line 6 of `CronValidator.java` throws an exception which is caught on Line 8. The error message includes the `value` string verbatim, and this reaches the sink on line 11 with no sanitization. Tracing such a flow across multiple files is a challenge for both humans and LLM agents.

Challenge 2: *Crafting an input that circumvents the branch conditions.* Even though there may be a *data flow* path from source to sink, this is only one of several execution paths that the program can take, depending on the *control flow*. In order to exercise this particular path, we have to call the `isValid` method with appropriate arguments such that the *control flow* reaches line 11 of `CronValidator.java`. The branch conditions are marked in green in Figure 1. We can see that there are 6 `if` conditions and 2 `try...except` blocks. Each of these corresponds to a constraint on the input that has to be satisfied in order for the control flow to reach the vulnerability sink. For example, the condition on lines 9-11 of `CronParser.java` expects the expression to have a certain number of fields (number of space-separated components). If there is a mismatch between `expressionLength` and the expected number of fields, `fields` will be `null`, and the control flow will be diverted down a non-vulnerable path. Note that this figure only captures a

portion of the full complexity of the program's control flow; for example, the `validate()` function on line 15 of `CronParser.java` also needs to fail and throw an `IllegalArgumentException`. This means that an input that triggers this vulnerability needs to be extremely carefully crafted to circumvent all of these branch conditions². This is representative of vulnerability-triggering inputs in general, as these tend to arise only in very specific edge cases.

Limitations of existing LLM agents: LLM agents tend to struggle with this kind of complex reasoning, and perform poorly out of the box. For instance, when we use CodeAct 2.1, a popular LLM agent, to generate a PoV test for the above vulnerability, it generates a test that calls `isValid()` with the following string as the `value` argument:

```
"${new java.io.FileWriter('/tmp/...').write('exploit')}
```

The intention is to inject Java code that writes to a file in the `tmp` directory. However, the agent fails to understand that the expression has to have a *specific number of fields* in order to get past the branch condition on lines 9-11 of `CronParser.java`. A valid Cron expression needs to have 6 or 7 space-separated components, which is what that branch condition checks internally. The generated expression has only 2 space-separated components, namely `"${new"` and `"java.io...}"`.

Our tool, FAULTLINE, uses a series of carefully crafted reasoning steps to guide LLM agents to address the above challenges and generate better PoV tests.

3 Methodology

In this section, we describe FAULTLINE, an LLM agent that automatically generates PoV test cases for a project with a reported vulnerability. In this paper, we focus on the following vulnerability types:

- **CWE-22 (Path Traversal):** This occurs when insufficient validation of user-supplied input in file path construction allows attackers to access files outside the intended directory using sequences such as `"../"`.
- **CWE-78 (OS Command Injection):** This enables attackers to execute arbitrary operating system commands by injecting malicious input into application-constructed system commands.
- **CWE-79 (Cross-Site Scripting):** This happens when unvalidated user input in web output allows attackers to execute malicious scripts in victims' browsers.
- **CWE-94 (Code Injection):** This allows attackers to inject and execute arbitrary code by exploiting insufficient input validation in code interpretation functions.

However, we emphasize FAULTLINE is not limited to these specific CWE categories, and our framework is general enough to permit extension to any other software vulnerability type.

Figure 2 describes the workflow of our agent. In the first stage (Section 3.1), we prompt the LLM to identify a source and a sink with a flow between them. In the second stage (Section 3.2), we leverage the agent to reason about the branch conditions encountered along the flow, and use these conditions to derive a set of conditions that an input has to satisfy in order to pass through this flow. In the final stage (Section 3.3), we use the information aggregated from previous stages to generate a PoV test case, and repair it based on build and

²<https://securitylab.github.com/advisories/GHSL-2020-212-cron-utils-ssti/>

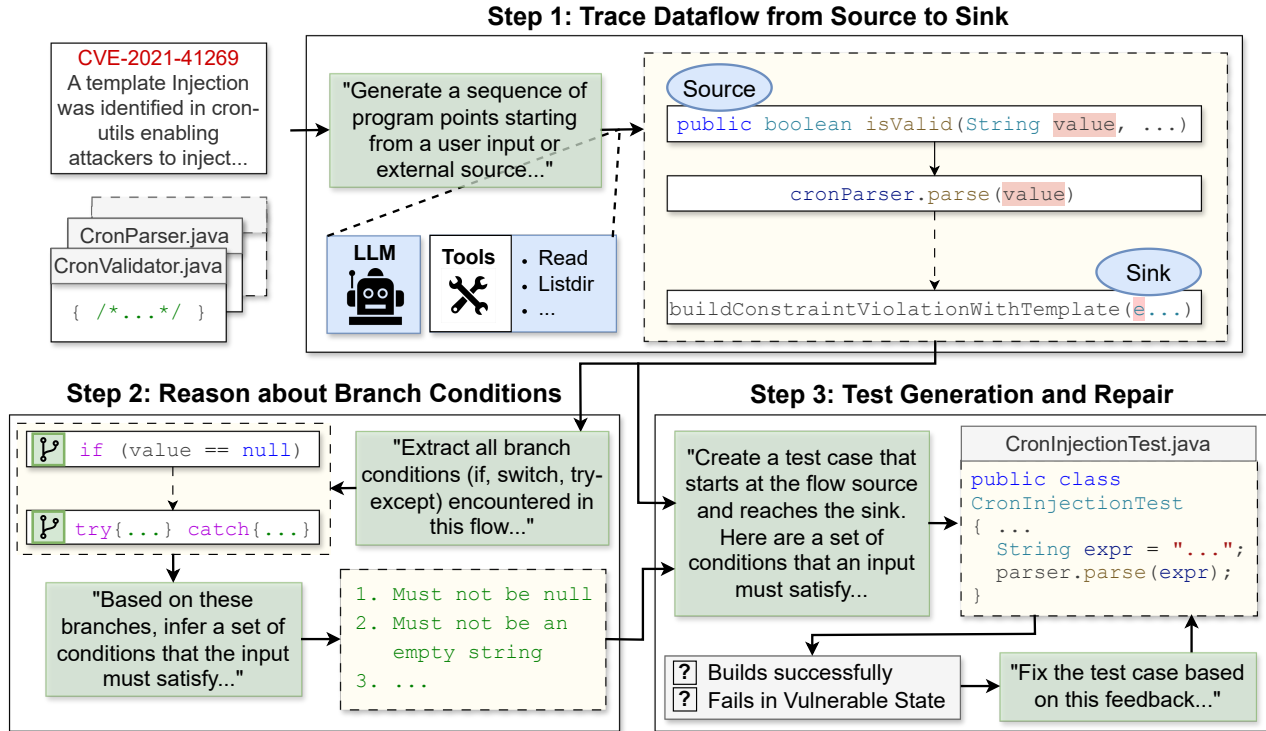


Figure 2: Our system FAULTLINE, for generating Proof of Vulnerability test cases.

execution feedback. Although the final reasoning generated in one stage is passed on to subsequent stages, the various stages do not share a common conversation memory. This keeps the length of each conversation in check. Full LLM prompts for all these stages are available in the appendix.

3.1 Data Flow Reasoning

We start with a project containing a known vulnerability, and the corresponding vulnerability report extracted from NVD database [24]. The first step towards generating a PoV test case is to process the report, scan the files in the repository, and understand the nature of the vulnerability. Often, the report provides scant information. This is sometimes done intentionally, so as to not inadvertently provide attackers with a blueprint on how to craft an exploit. However, every report includes at least a) the CWE categorization of the vulnerability, b) affected versions of the project, c) mitigation strategies for developers. Figure 3 shows an example of a vulnerability report corresponding to the motivating example in Section 2.2.

Although this report is vague, it provides hints that can be used to deconstruct the vulnerable flow. The report states that the vulnerability allows attackers to "inject arbitrary Java EL expressions". This narrows the search for *sinks* to program constructs that can process Java EL expressions. Further, the report indicates that the vulnerability arises from "using the @Cron annotation to validate untrusted Cron expressions". This means that the *source* is likely to be an API that accepts a Cron expression from a user.

Given these semantic hints from the vulnerability report, the next step is to automatically identify the data flow from source

CWE-94: Improper Control of Generation of Code ('Code Injection')

Details: cron-utils is a Java library to define, parse, validate, migrate crons as well as get human readable descriptions for them. In affected versions A template Injection was identified in cron-utils enabling attackers to inject arbitrary Java EL expressions, leading to unauthenticated Remote Code Execution (RCE) vulnerability. Versions up to 9.1.2 are susceptible to this vulnerability. Please note, that only projects using the @Cron annotation to validate untrusted Cron expressions are affected. The issue was patched and a new version was released. Please upgrade to version 9.1.6. There are no known workarounds.

Figure 3: An example of a vulnerability report, for CVE-2021-41269. Although it is vague, the highlighted sections provide some hints about the sources and sinks of this vulnerability.

to sink within the codebase. One natural approach would be to leverage existing static analysis tools designed for this purpose. CodeQL [1], for instance, is an industry standard tool used for detecting dangerous flows between sinks and sources. However, several limitations make it unsuitable for our use case: a) it cannot directly utilize semantic information from the report to guide its detection algorithm, b) it requires a non-trivial amount of effort to set up and run for each project and programming language, c) it is tuned for high precision, causing it to miss several flows. In fact, CodeQL fails to detect the flow corresponding to our motivating example [20]!

For all of the above reasons, we opt to use an LLM acting as an autonomous agent to reconstruct the vulnerable flow. We include the entire vulnerability report in the initial prompt, and instruct the LLM agent to use the information in the report to generate a flow starting from a source and reaching a sink. To enable the agent to explore the project’s source code, we provide it with tools to list a directory (`ListDir`) and read a file (`Read`). In order to use semantic hints from the vulnerability report, such as the `@Cron` annotation, we additionally equip the agent with tools to search for files by name (`Find`), and to search for files containing specific strings (`Grep`).

The output of this stage is a sequence of program points comprising a flow, where each point is identified by a short snippet of code (1-2 lines). Each point is additionally labeled with:

- (1) the name of the file it is contained in,
- (2) the name of the variable that carries the vulnerable flow, and
- (3) a role, *i.e.*, Source, Sink, or Intermediate Node.

As an example, here is the portion of the data flow reasoning output corresponding to the *source* of the flow in Figure 1.

```
{ "code": "public boolean isValid(String
    value, ...",
  "role": "Source",
  "variable": "value",
  "file": ".../CronValidator.java" }
```

The entire flow output consists of a sequence of points in the above format, starting with a source and ending with a sink. We use this flow as an input for our subsequent reasoning and generation steps.

3.2 Control Flow Reasoning

Once a flow from source to sink has been identified, the next step towards generating a PoV test case is reasoning about how to generate an input that will actually traverse this flow. We decompose this task into two steps - extracting branch conditions, and reasoning about conditions on the input.

Extracting branch conditions. We collect the flow reasoning generated by Step 1, and instruct an LLM agent to follow this flow and extract all the branch conditions that an input might encounter on the way. This includes not just `if-else` and `switch` constructs, but also `try-except` blocks. Each branch represents a potential opportunity for the control flow to be diverted down a non-vulnerable path that never reaches the sink, or pass through program points that sanitize the flow and render it non-threatening. Reasoning about every single branch, therefore, is crucial to constructing an input that can reach the sink. We refer to the path represented by the program points corresponding to these branch conditions as the *control flow path*.

Note that the control flow path can be *very different* from the data flow path (Section 3.1). For example, in Figure 1, the data flow (shown in red) spans the two files `CronValidator.java` and `CronParser.java`. However, the control flow path includes a completely different set of nodes (shown in green). Further, a portion of the control flow path traverses a file not shown in this figure, `SingleCron.java`. This is induced by the call to

`SingleCron::validate()` on line 14 of `CronParser.java`, which must throw an `IllegalArgumentException` in order for the program to proceed down the path towards the sink. So although one might imagine the data flow and control flow paths to be similar, they have many fundamental qualitative differences.

Similarly to Section 3.1, we equip the LLM agent with `ListDir`, `Read`, `Find` and `Grep` tools. We prompt it to extract each branch condition as a short snippet of code (1-2 lines). Each condition is required to be additionally labeled with:

- (1) its type (If-Else, Switch, etc.),
- (2) the name of the file it is contained in, and
- (3) the desired outcome of the branch, in order for the input to proceed down a path that leads to the sink.

For example, the portion of the branch reasoning output corresponding to the branch condition in line 3 of `CronValidator.java` in Figure 1 is:

```
{ "code": "if (value == null)",
  "type": "If-Else",
  "file": ".../CronValidator.java",
  "outcome": "False - the value should not
    be null" }
```

Reasoning about conditions on the input. Analyzing these branch conditions carefully can provide information on how to generate an appropriate test input that walks a metaphorical tightrope, traversing the correct path through these branches. However, we found that LLMs often struggled to parse these branch conditions and produced tests with inadequate inputs. To make the connection between branch conditions and input requirements more explicit, we add a further reasoning step.

We ask the agent to reflect on its own output and infer a set of conditions that an external input must satisfy, in order to pass through all the branches. This is essentially a compositional reasoning task, in which the agent must aggregate information from each branch, and compose them into a set of unifying input conditions. For illustrative purposes, here are some of the conditions generated by the agent for our motivating example from Figure 1 and Section 2.2:

1. The input must not be null...
2. The input must not be an empty string after trimming whitespace...
3. The input must not contain || ...

Since these conditions succinctly summarize the detailed branch information produced earlier, we collect these conditions for use in the next step of the tool and *discard* the detailed branch information.

3.3 Test Generation and Repair

At this stage, we have a detailed description of a flow from source to sink, along with a series of constraints that a test input must satisfy in order to exercise this flow. The final stage of our system involves using the flow information and input constraints

to generate an initial proof-of-vulnerability test case, followed by feedback-driven repair.

Criteria for Success To construct a framework that can generate a PoV test, we first have to understand what it means for a PoV test to be successful. This is surprisingly non-trivial. Consider our motivating example again (Figure 1). The root cause of the vulnerability is that the error message can, in certain cases, reproduce the user-provided `value` string verbatim. A hypothetical PoV test could call `isValid` with a `value` string embedded with a specific sequence of characters, say `"abcd"`, in the appropriate format to reach the sink. It could then assert that the error message does not contain `"abcd"`. Prima-facie, this would satisfy the requirements of a PoV test:

- (1) It would *fail* if and only if the vulnerability *exists*,
- (2) It would execute the vulnerable method and use the observed result directly in an assertion to check the presence of a bug.

However, this is a *Code Injection* vulnerability, and the would-be PoV test *does not actually inject any code*! This shows us that the definition of a successful PoV test must necessarily be tailored to the type of vulnerability, e.g., a Code Injection vulnerability must inject code and execute it in addition to satisfying the above three conditions.

Previous works on exploit generation [2, 28] have attempted to group exploits based on their CWE category and develop criteria for each category that can be automatically checked. For example, an exploit for OS Command Injection (CWE-78) could be verified by checking whether the test can execute a specified binary like `/usr/bin/mybin`. These are akin to Capture-The-Flag (CTF) challenges. However, these definitions can be too restrictive in our setting. For example, [CVE-2014-3576](#) is an OS Command Injection vulnerability that allows an attacker to specifically execute the `shutdown` command. A definition based on `/usr/bin/mybin` would be a mismatch for this exploit.

We instead opt to use semantic criteria that can be manually checked. For each of the CWE categories we consider in this paper, we define what it means for an exploit of that category to be successful:

- **CWE-22 (Path Traversal)**: The test case must use a public API of the project to read from or write to at least one file outside the project directory.
- **CWE-78 (OS Command Injection)**: The test case must use a public API of the project to execute any shell command that is not intended by the application.
- **CWE-79 (Cross-Site Scripting)**: The test case must call a public API of the project with an input that contains embedded scripting code, and show that this input is not sanitized properly.
- **CWE-94 (Code Injection)**: The test case must call a public API of the project with an input that contains embedded code, and this code must be executed.

Generating an initial test: When humans write code to solve a task, we rarely write correct functional code in a single attempt. Instead, we usually follow an iterative process, where we write some code, observe its behavior, add debugging statements if necessary, re-run the code, and so on. We used this process flow as a guide while developing our test generation component. In addition to the tools mentioned in previous stages, i.e., `ListDir`, `Read`, `Find` and

Create a test case that FAILS (exits with non-zero code) if the vulnerability EXISTS, and PASSES (exits with code 0) if the vulnerability DOES NOT EXIST.

This is a Code Injection vulnerability (CWE-94). The test case must call a public API of the project with an input that contains embedded code, and this code must be executed. This test should actually run the vulnerable code in the project.

- It should NOT read the source code to check for the presence of a vulnerability.

- It should NOT "simulate" the vulnerability by running some separate code that does not use the project.

Here is a flow consisting of a sequence of program points to reach the vulnerability:

```
{flow}
```

The test should start from the flow 'source' and reach the 'sink'. It should be designed such that it passes through all the branch conditions on the way. This means that the input and method calls should be carefully crafted, satisfying the following conditions:

```
{input_conditions}
```

Figure 4: A portion of our test generation prompt for a CWE-94 (Code Injection) vulnerability. The highlighted portion is modified depending on the CWE category. For the full prompt, refer to the appendix.

`Grep`, we also give the agent the ability to write to files (`Write`) and run the current test code to observe its output (`Run`).

To ensure consistent environment-independent execution, we ask the agent to wrap each project and its dependencies as a Docker container. The container, when built and run, must execute the test case. The `Run` tool does not permit the execution of arbitrary terminal commands; rather, it just builds and runs the Docker container, and furnishes the agent with the output of these commands.

Figure 4 shows a portion of our test generation prompt for a CWE-94 (Code Injection) vulnerability. We include the flow information and input constraints obtained in Section 3.1 and Section 3.2 respectively. The prompt is specific to the CWE category of the vulnerability and includes the criteria for an exploit to be considered successful. We also add instructions in the prompt to avert certain common failure modes — 1) tests which don't actually run the program, and instead match shallow patterns in the source code to check for the presence of certain text, 2) tests that simulate the vulnerability by re-implementing a simplified version of it, without running the existing project code. We explicitly instruct the agent to avoid these patterns of behavior.

Once the agent has generated a test and is satisfied that it runs correctly, we instruct it to respond `<DONE>` to trigger the next phase. **Feedback-driven repair:** We validate the generated test case with two automated checks. We first build the project as a Docker image, and check if it completes successfully. If it does, we run a container with the built image, and check that the run fails (exits with non-zero code). If either of these stages does not complete as expected, i.e., if the build fails or the run succeeds, we collect the output from that stage and use it as feedback for the agent.

| | CWE-Bench-Java | PrimeVul | Total |
|----------------------|----------------|-----------|------------|
| Path Traversal | 35 | 13 | 48 |
| Command Injection | 6 | 4 | 16 |
| Cross-Site Scripting | 15 | 3 | 18 |
| Code Injection | 14 | 10 | 24 |
| Total | 70 | 30 | 100 |

Table 1: The statistics of our benchmark datasets, showing the number of instances of each vulnerability type.

We prompt the agent with this feedback, instructing it to fix the test by carefully analyzing errors or messages in the output, and reasoning step by step about what might have gone wrong. We perform this feedback-driven repair in a loop until a preset maximum number of iterations is reached.

4 Experimental Setup

4.1 Benchmarks

CWE-Bench-Java [20] is a dataset of 120 Java programs with known vulnerabilities. The vulnerabilities span 4 CWE categories – Path Traversal, OS Command Injection, Cross-Site Scripting (XSS) and Code Injection. Each vulnerability includes metadata like its vulnerability report from the National Vulnerability Database [24], the URL of the GitHub repo of the project, buggy and fixed commit hashes, build instructions, and the classes and methods changed to fix the vulnerability. We attempted to clone each project and build it in both the vulnerable and fixed states, using the build scripts provided with the CWE-Bench-Java dataset. We were unable to fetch the fix commit or build at the fixed commit for some of these projects, leaving us with a filtered dataset of 70 Java programs.

PrimeVul [11] is a dataset of over 7000 vulnerabilities in C and C++ programs. We filter these to include only vulnerabilities belonging to our selected CVE types, and select 30 vulnerabilities at random from the filtered set. However, PrimeVul does not annotate each project with build information. So this is possibly a more challenging setting for PoV test generation, where the model has to build the project successfully as a prerequisite for generating a test.

Our entire evaluation dataset therefore consists of 100 problem instances (70 from CWE-Bench-Java and 30 from PrimeVul) covering 3 programming languages (Java, C and C++).

4.2 Implementation

Setting up benchmarks: We reset each project to the vulnerable commit and build it as a Docker container. We make sure that the Dockerfile contains all the dependencies needed for the project, and does not require any external volumes. This ensures that a) the tests are run in a sandboxed environment, and b) the runs are reliably reproducible.

Implementing the system: FAULTLINE is implemented in Python and run with Docker. We developed custom interfaces for all the tools we discussed in Section 3. The underlying LLM for all our agent calls is Claude-3.7-Sonnet, which we access through [litellm](#).

We set a maximum budget of 5 USD and a time budget of 40 mins for each project.

4.3 Baselines

Our main baseline is the CodeAct 2.1 agent [29] running in the OpenHands framework [31]. This is a general-purpose, open-source, software agent. Although there are many other LLM agents that are specialized for the task of fixing software bugs, fixing bugs is a *very different problem* from generating tests. CodeAct 2.1 with OpenHands performs competitively with other specialized test generation models [25] on the SWE-Bench benchmark [18], and its capabilities extend to any general software engineering task. It outperforms popular agents like SWE-Agent by a large margin [30]. So it is a natural choice to use as a benchmark. Just as in FAULTLINE, we use Claude 3.7 Sonnet as the underlying LLM and impose the same budget and time constraints. The full prompt that we used for OpenHands is available in the appendix.

4.4 Metrics

We follow a set of steps to evaluate the correctness of a PoV test. If any of these steps fails, we abort the evaluation and return failure. In order:

- (1) **Build:** Build the project along with its created tests at the vulnerable commit, as a Docker image.
- (2) **Run:** Run the Docker image as a container, and check that it exits with non-zero code.
- (3) **Check coverage:** Check that the program flow in the previous step reached a method corresponding to the vulnerability. To evaluate this, we instrument each function to print its name when it is called. If the output contains the name of any method that was changed as part of the vulnerability fix, then we consider this satisfactory. This helps weed out tests that use shallow pattern matching against the text of the source program.

If a test passes all these 3 criteria, then the final step is to **manually inspect** it to evaluate whether it satisfies the category-specific criteria listed in Section 3.3. For example, if the vulnerability is Code Injection, then we verify that the test calls a public API of the project with an input that contains embedded code, and that the code is actually executed. If the test passes this check, we consider it correct.

5 Experimental Results

We evaluate our approach through the following research questions:

- **RQ1: Performance of our tool.** How many PoV tests is FAULTLINE successfully able to generate? How does this compare with our baselines?
- **RQ2: Different vulnerability types.** How does the performance of FAULTLINE vary across vulnerability types as represented by CWE categories?
- **RQ3: Evaluating our design choices.** What is the impact of the flow reasoning and branch reasoning components on the performance of FAULTLINE?

5.1 RQ1: Performance Evaluation

In this section, we measure the effectiveness of FAULTLINE in generating PoV tests, and measure its performance relative to our baselines.

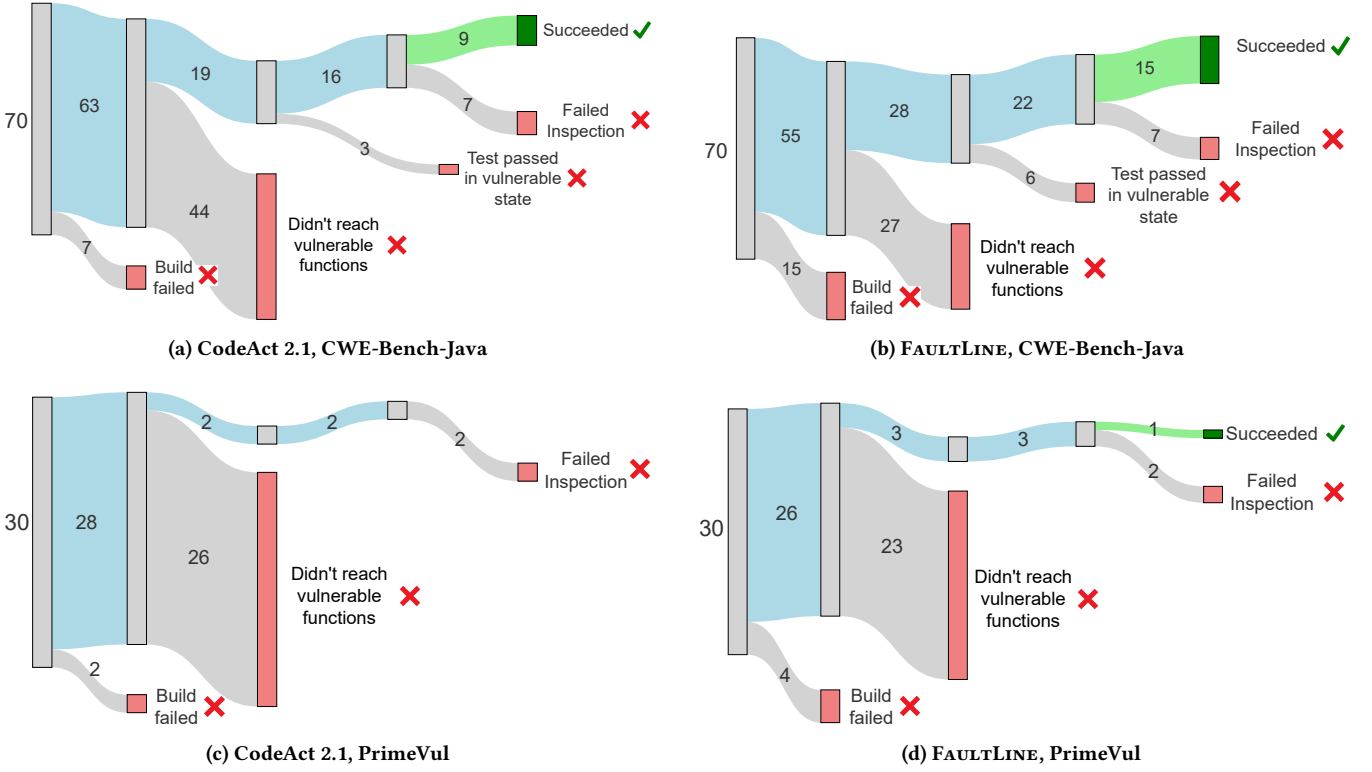


Figure 5: A detailed analysis of the test generation performance of CodeAct 2.1 (on the left) vs FAULTLINE (on the right), for our two benchmarks CWE-Bench-Java and PrimeVul. FAULTLINE is able to generate more successful tests (15 vs 9 and 1 vs 0), as well as more tests that reach the vulnerable functions (28 vs 19 and 3 vs 2).

Evaluation: We run FAULTLINE on our benchmark dataset of 100 programs. We also run CodeAct 2.1 in the same setting, with identical time and budget constraints. For each generated test case, we evaluate it according to the process defined in Section 4.4. If a test fails at a particular stage of this checking process, we collect that information.

Discussion: The overall results are in Figure 5. We make the following observations:

- **CWE-Bench-Java results:** FAULTLINE is able to generate successful PoV tests for 15 out of 70 CWE-Bench-Java projects (21%) compared to CodeAct which succeeds for only 9 (13%). This is a significant gap, and clearly shows the benefits of our multi-stage agentic workflow.
- **PrimeVul results:** On the PrimeVul dataset, CodeAct is unable to solve any of the 30 problems, whereas FAULTLINE is able to solve 1. As mentioned in Section 4, PrimeVul does not have build scripts for individual projects; however this (perhaps surprisingly) does not seem to pose a challenge to either CodeAct or FAULTLINE. Out of 30 projects, the resulting Dockerfiles build successfully for 27 and 26 of the projects, for CodeAct and FAULTLINE respectively. However, the test flow reaches the vulnerability for only 3 and 2 projects respectively. This suggests that test generation is particularly challenging on PrimeVul not because of a lack of build information, but because of a lack of understanding of the flow of each vulnerability. Further research is needed to develop better PoV test generation tools for these projects.

- **Vulnerable function coverage:** FAULTLINE is able to generate tests that reach vulnerable functions for 28 CWE-Bench-Java projects versus only 17 for CodeAct, and likewise for PrimeVul (3 vs 2). This is evidence that our flow and branch reasoning steps are having the desired effect, enhancing the ability of the agent to produce test inputs that reach vulnerability sinks.

Summary: FAULTLINE is able to generate PoV tests for 16 vulnerabilities, as compared to just 9 for the CodeAct 2.1 baseline. Further, FAULTLINE-generated tests reach the vulnerable functions in 31 projects, as compared to 19 for the baseline.

5.2 RQ2: Vulnerability Types

In this section, we evaluate how the performance of our agent varies across vulnerability types.

Evaluation: We collect the results of PoV test generation as described in Section 5.1, and categorize each test according to the CWE-ID of the project it corresponds to. We plot the percentage of successful tests in each category, for both tools.

Discussion: The results are shown in Figure 6. We make the following observations:

- As measured by average percentage of successful PoV tests generated, CWE-94 (Code Injection) is the hardest category, with both tools reporting a success rate of just 8%. CWE-22 (Path Traversal) and CWE-79 (Cross-Site Scripting) seem to be relatively easier by

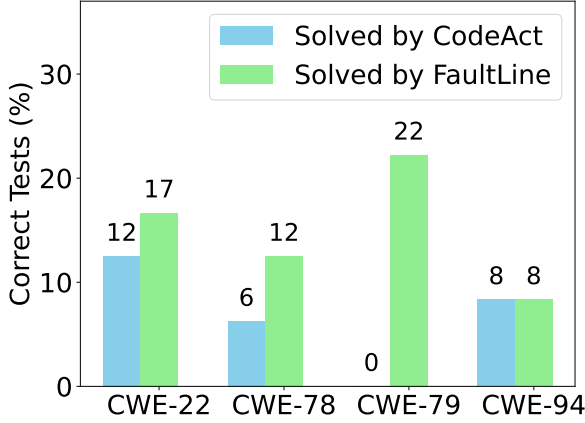


Figure 6: Percentage of correct PoV tests per CWE category.

| Successful PoV Tests | |
|----------------------|----|
| w/o Flow, w/o Branch | 11 |
| w/o Flow | 9 |
| w/o Branch | 11 |
| FAULTLINE | 16 |

Table 2: Assessing the impact of the flow and branch reasoning steps.

comparison, but overall it is clear that this is a challenging task irrespective of category.

- Another observation is that for each category, FAULTLINE is consistently on par with or better than CodeAct 2.1. This shows that its performance gains are not specific to any one type of vulnerability. Although in this paper we have only evaluated our technique on 4 CWE categories, based on these results we expect it to generalize well to other categories too.

Summary: FAULTLINE performs on par or improves over the baseline for all 4 CWE categories, showing its generalizability.

5.3 RQ3: Evaluating Design Choices

In this section, we evaluate the impact of the flow reasoning and branch reasoning components on the performance of FAULTLINE.

Evaluation: We design three configurations of the tool by removing flow analysis, branch analysis, and both flow and branch analysis. Running the each of these configurations on all 100 examples was prohibitively expensive, so instead, we selected the 16 projects corresponding to tests generated by FAULTLINE that passed evaluation, and ran each tool configuration on this subset of 16 projects.

Discussion: The results are shown in Table 2. We can see that FAULTLINE is the best performing configuration, and that the configurations without flow and branch analysis do not generate as many successful tests. This shows that each component of our system is crucial to its overall performance. One interesting observation is that branch reasoning alone, without flow reasoning, performs worse than a configuration with neither branch nor flow

```

<FLOW>
{"role": "Source",
 "code": "public boolean isValid(String value,
 ConstraintValidatorContext context) {",
 "variable": "value",
 "file": "...CronValidator.java",
 "remarks": "The entry point where untrusted cron
 expressions are received for validation"
}
...
{"role": "Sink",
 "code": "context.buildConstraintViolationWithTemplate(
 e.getMessage()).addConstraintViolation();",
 "variable": "e.getMessage()",
 "file": "...CronValidator.java",
 "remarks": "The unvalidated input is used in a template
 context, allowing for Java EL expression injection"
}
</FLOW>
<CONDITIONS>
...
6. The input must have a valid number of parts that matches
one of the expected cron expression formats
7. The input must contain at least one special character (/,
-, or ,) to trigger the complex parsing path
...
</CONDITIONS>
    
```

Figure 7: A portion of the output of flow and branch reasoning of FAULTLINE for the motivating example from Section 2.2. Deriving the right conditions on the input string allows the test-generation agent to generate a PoV test that reaches the vulnerability sink.

reasoning. In this configuration, we are essentially prompting the model to extract branch conditions along a unspecified data flow, and further, reason about conditions that an unspecified source input has to satisfy. This requires two reasoning steps, and intuitively, scaffolding the reasoning process should yield better results.

To further investigate the effectiveness of our multi-step reasoning pipeline, we inspect the system’s outputs for our motivating example from Section 2.2. The system generates the correct flow from source to sink, shown in the top half of Figure 7. Then, using this flow, FAULTLINE extracts branch conditions and generates input constraints, a portion of which are shown in the bottom half of Figure 7.

Notice that this contains the constraint “the input must have a valid number of parts”. As we discussed in Section 2.2, CodeAct fails to understand this requirement and generates an input containing just the wrong number of space-separated components. On the other hand, FAULTLINE is able to use these constraints to generate a test that calls `isValid` with the following string: `"* * * * * ${T(java.lang.Runtime).getRuntime().exec('touch /tmp/abc')}"`. This has 7 space-separated components, which is a valid format and allows the dangerous input to reach the vulnerability sink without sanitization.

Summary: Without its flow or branch reasoning components, FAULTLINE solves between 9 and 11 problems, as compared to 16 in its full configuration. This shows the utility of each component.

6 Related Work

Vulnerability Datasets: There are many curated vulnerability datasets in different languages, some with proof-of-vulnerability tests. BigVul [12] is a dataset of 3,754 C/C++ vulnerabilities, and PrimeVul [11] builds on BigVul to create a higher quality dataset of 6,968 C/C++ vulnerabilities. CrossVul [26] has ~13,000 vulnerabilities in 40 programming languages. SVEN [15] consists of ~1600 C/C++/Python vulnerabilities. These datasets include, for each vulnerability, the URL of the GitHub repository from which the vulnerability was sourced, the commit message corresponding to the fix, and the patch. However, these datasets lack information on how to build each project and reproduce the vulnerability. They also lack proof-of-vulnerability test cases.

CWE-Bench-Java [20] is a dataset of 120 Java vulnerabilities with build information for each project, but no proof-of-vulnerability test cases. Vul4J [9] is a small dataset of 79 Java vulnerabilities, each of which is reproducible and has proof-of-vulnerability test cases. ARVO [21] is a continually expanding database of C/C++ vulnerabilities collected from Google’s OSS-Fuzz tool. SecBench.js [7] is a dataset of JavaScript vulnerabilities, some of which have exploit code.

LLM Agents for Bug Reproduction: There is a line of research on generating bug reproduction tests from reports. LIBRO [19] frames this as a few-shot code generation problem, where an LLM is shown examples of bug reports and corresponding tests, before being asked to generate a test for a given report. Cheng et al. [10] build on the LIBRO framework by designing an agentic system with a fine-tuned code editing tool. Otter [3, 4] is an LLM agent workflow that uses systematic reasoning to generate bug reproduction tests. However, this is qualitatively very different from vulnerability exploit generation, which involves generating carefully crafted inputs that can traverse long sequences of method calls within a program.

Proof-of-Vulnerability Test Generation: There is a long line of work that predates LLMs [5, 6, 8, 16, 17], based on deriving constraints on a program’s input and using symbolic execution to solve these constraints. However, these are specialized to certain kinds of vulnerabilities, and cannot reach vulnerabilities that are deeply embedded in a program. SemFuzz [33] is a fuzzing tool that uses semantic information from vulnerability reports to perform guided fuzzing. However, this can only generate relatively simple kinds of inputs, and can only detect vulnerabilities that result in runtime errors like crashes, resource leaks or infinite loops. ARVO [21] collects bug reports from Google’s OSS-Fuzz [27], derives build and dependency information for each project, and creates a dataset of reproducible vulnerabilities. However, ARVO relies on the bug report already containing the exact input that triggers the vulnerability.

There has been research on using LLM agents to generate exploits for web applications in a one-day [13] and zero-day [36] setting. Zhu et al. [35] design a comprehensive benchmark for evaluating exploit generation in web applications. However, the setting

of these works is significantly different from ours, because exploiting web vulnerabilities requires interacting with a webpage, e.g., entering text in a box or clicking a button; as opposed to our setting which requires writing code that calls an API. EniGMA [2] augments an SWE-Agent [32] with additional tools like an interactive debugger, to enable it to solve Capture-The-Flag (CTF) problems. The solution to a CTF problem is a string, or “flag”, that has to be retrieved. This is qualitatively very different from our setting, which involves writing test cases.

PoCGen [28] is a concurrent work to ours that involves generating proof-of-concept exploits for vulnerabilities in NPM packages. However, it relies on static and dynamic analysis tools for NPM, which limits its generalizability across programming languages.

7 Limitations and Threats

As with any experimental study, the conclusions of our work must be considered in the context of the following potential threats to validity.

Data leakage: The knowledge cutoff date of Claude-3.7-Sonnet, our base LLM, is November 2024. Our data sources for vulnerabilities, CWE-Bench-Java and PrimeVul, were curated before this date. It is very likely that the LLM has seen several of these vulnerabilities as part of its training data. Therefore, it is possible that our careful prompting is not actually *eliciting reasoning*, but simply enabling the model to *recall instances* from its training data. This threat is somewhat mitigated by two factors:

(1) PoV tests are seldom made public for security-related reasons. Although the model may have seen several of the vulnerabilities in its training data, it is unlikely to have seen the corresponding PoV tests.

(2) Our primary comparison is with the CodeAct agent, which uses the same underlying LLM. Therefore, there is no unfair advantage gained by our agentic framework compared to the baseline.

Test success in fixed state: An ideal proof of vulnerability test must not only fail when the vulnerability exists, but also *pass* when it is fixed. However, our evaluation criteria do not include a check that the test passes in the fixed state. When we tried running our tests on the fixed versions of each project, we observed certain issues:

(1) Sometimes, a test fails to build when the project is in the fixed state, because of mismatched dependencies or versions. This is not the agent’s fault, because it does not have access to the fixed state of the project.

(2) There are also certain cases where the test exits with a non-zero code in the fixed version because the project detects the attempted exploit and raises an exception. The test should ideally catch this exception and return success, but once again, the agent does not have access to the source code of the fixed version of the project. So it cannot know, a priori, the exact exception that will be thrown, or even the fact that an exception will be thrown at all.

For all of these reasons, we choose not to include passing in the fixed state as a criterion for a successful PoV test, and we defer the question of how to accomplish this to future work.

Manual Inspection: The final step of our evaluation procedure (Section 4.4) is a manual inspection. Although this is based on objectively verifiable criteria, there is a possibility of errors in human

judgment during the labeling process. This is somewhat mitigated by the fact that the number of examples necessitating such manual labeling is low, and we made every effort to be thorough with our analysis, but nevertheless it remains a limitation of our experimental design.

Generalizability: Finally, we acknowledge that FAULTLINE is evaluated on only 4 CWE categories, which means that our conclusions have to be contextualized accordingly. However, our system design does not make any assumptions on the type of vulnerability, and we see consistent gains over the baseline across all our 4 categories. Thus, we expect that the conclusions would hold for other categories too, but we defer this investigation to future work.

8 Conclusion

Proof-of-vulnerability tests are of vital importance to enable developers to understand a vulnerability and avoid regressions. Generating these tests involves subtle reasoning about program properties, and proves extremely challenging for state-of-the-art LLM agents. In this paper, we have developed FAULTLINE, a system that utilizes carefully designed LLM reasoning steps to automatically generate PoV tests. Our results highlight the effectiveness of multi-step reasoning workflows in LLM agents, and our benchmark of 100 projects represents a challenging direction for further research in LLM agents and test generation.

References

- [1] [n. d.]. <https://codeql.github.com/>
- [2] Talor Abramovich, Meet Udeshi, Minghao Shao, Kilian Lieret, Haoran Xi, Kimberley Milner, Sofija Jancheska, John Yang, Carlos E Jimenez, Farshad Khorrami, et al. [n. d.]. EnGMA: Interactive Tools Substantially Assist LM Agents in Finding Security Vulnerabilities. In *Forty-second International Conference on Machine Learning*.
- [3] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: Generating Tests from Issues to Validate SWE Patches. *arXiv preprint arXiv:2502.05368* (2025).
- [4] Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. 2024. TDD-Bench Verified: Can LLMs Generate Tests for Issues Before They Get Resolved? *arXiv preprint arXiv:2412.02883* (2024).
- [5] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 641–652.
- [6] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.
- [7] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An executable security benchmark suite for server-side JavaScript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1059–1070.
- [8] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 143–157.
- [9] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. 2022. Vul4j: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 464–468.
- [10] Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambroner, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. 2025. Agentic Bug Reproduction for Effective Automated Program Repair at Google. *arXiv preprint arXiv:2502.01821* (2025).
- [11] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624* (2024).
- [12] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th international conference on mining software repositories*. 508–512.
- [13] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. 2024. Llm agents can autonomously exploit one-day vulnerabilities. *arXiv preprint arXiv:2404.08144* 13 (2024), 14.
- [14] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming, Nachiappan Nagappan, et al. 2023. A survey on automated software vulnerability detection using machine learning and deep learning. *arXiv preprint arXiv:2306.11673* (2023).
- [15] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.
- [16] Hong Hu, Zheng Leong Chua, Sendriu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of {Data-Oriented} Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*. 177–192.
- [17] Shih-Kun Huang, Han-Lin Lu, Wai-Meng Leong, and Huan Liu. 2013. Craxweb: Automatic web application testing and attack generation. In *2013 IEEE 7th International Conference on Software Security and Reliability*. IEEE, 208–217.
- [18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).
- [19] Sungmin Kang, Juyeon Yoon, Nargiz Askarbekkyzy, and Shin Yoo. 2024. Evaluating diverse large language models for automatic and general bug reproduction. *IEEE Transactions on Software Engineering* (2024).
- [20] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*.
- [21] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupe, Hammond Pearce, Brendan Dolan-Gavitt, et al. 2024. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software. *arXiv preprint arXiv:2408.02153* (2024).
- [22] MITRE Corporation. 2025. Common Weakness Enumeration. <https://cwe.mitre.org> Accessed: July 18, 2025.
- [23] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 919–936.
- [24] National Institute of Standards and Technology. 2025. National Vulnerability Database. <https://nvd.nist.gov/vuln> Accessed: July 15, 2025.
- [25] Graham Neubig and Xingyao Wang. 2024. OpenHands CodeAct 2.1: An Open, State-of-the-Art Software Development Agent. *All Hands AI Blog* (1 November 2024). <https://www.all-hands.dev/blog/openhands-codeact-21-an-open-state-of-the-art-software-development-agent>
- [26] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1565–1569.
- [27] Kostya Serebryany. 2017. {OSS-Fuzz}-Google’s continuous fuzzing service for open source software. (2017).
- [28] Deniz Simsek, Aryaz Eghbali, and Michael Pradel. 2025. PoCGen: Generating Proof-of-Concept Exploits for Vulnerabilities in Npm Packages. *arXiv preprint arXiv:2506.04962* (2025).
- [29] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.
- [30] Xingyao Wang, Bowen Li, and Graham Neubig. 2024. Introducing OpenDevin CodeAct 1.0, a new State-of-the-art in Coding Agents. *Blog* (7 May 2024). <https://xwang.dev/blog/2024/opendevin-codeact-1.0-swebench/>
- [31] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=OJd3ayDDoF>
- [32] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.
- [33] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2139–2154.
- [34] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.
- [35] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Bunn, et al. 2025. CVE-Bench: A Benchmark for AI Agents’ Ability to Exploit Real-World Web Application

- Vulnerabilities. *arXiv preprint arXiv:2503.17332* (2025).
- [36] Yuxuan Zhu, Antony Kellermann, Akul Gupta, Philip Li, Richard Fang, Rohan Bindu, and Daniel Kang. 2024. Teams of llm agents can exploit zero-day vulnerabilities. *arXiv preprint arXiv:2406.01637* (2024).

A Prompts

Listing 1: System Prompt

You are a helpful AI assistant that can interact with a computer to solve tasks.

<ROLE>

Your primary role is to assist users by executing commands, modifying code, and solving technical problems effectively. You should be thorough, methodical, and prioritize quality over speed.

Your code will never be read by humans, so focus on correctness, not style.

</ROLE>

<EFFICIENCY>

- * Each action you take is somewhat expensive. Minimize unnecessary actions.
- * When exploring the codebase, use the find and grep tools with appropriate filters to minimize unnecessary operations.
- * You do not have access to the internet, so do not attempt to search online for information.

</EFFICIENCY>

<CODE_QUALITY>

- * Write clean, efficient code with minimal comments. Avoid redundancy in comments: Do not repeat information that can be easily inferred from the code itself.
- * When implementing solutions, focus on making the minimal changes needed to solve the problem.
- * Before implementing any changes, first thoroughly understand the codebase through exploration.
- * If you are adding a lot of code to a function or file, consider splitting the function or file into smaller pieces when appropriate.

</CODE_QUALITY>

<PROBLEM_SOLVING_WORKFLOW>

1. EXPLORATION: Thoroughly explore relevant files and understand the context before proposing solutions
2. ANALYSIS: Consider multiple approaches and select the most promising one
3. IMPLEMENTATION: Make focused, minimal changes to address the problem

</PROBLEM_SOLVING_WORKFLOW>

<TROUBLESHOOTING>

- * If you've made repeated attempts to solve a problem but tests still fail or the user reports it's still broken:
 1. Step back and reflect on 5-7 different possible sources of the problem
 2. Assess the likelihood of each possible cause
 3. Methodically address the most likely causes, starting with the highest probability
 4. Document your reasoning process

</TROUBLESHOOTING>

Listing 2: Flow Reasoning

The project I am working with has a vulnerability, reported as a CWE. The issue description says:

{description}

You do not have access to the internet or GitHub to look up more details.

There are no vulnerability reports in the project directory either.

{tool_description}

Could you generate a sequence of program points to reach the vulnerable point (sink), starting from an external input (source)? This corresponds to a vulnerable "flow" through the program.

The flow should take the form of a sequence of program points, each in the following format:

```
{
  "role": "Source|Intermediate|Sink",
  "code": "Source code of program point (1-2 lines)",
  "variable": "Variable name",
  "file": "File path (absolute)",
  "remarks": "Comments about this point, if any"
}
```

You can use multiple intermediate steps and tool invocations, but when you are finished, your final response should contain the flow in the above format, within the tags <FLOW> and </FLOW>.

Listing 3: Branch Reasoning Part 1

The project I am working with has a vulnerability, reported as a CWE. The issue description says:

{description}

You do not have access to the internet or GitHub to look up more details.

There are no vulnerability reports in the project directory either.

{tool_description}

Here is a flow consisting of a sequence of program points to reach the vulnerability:

{flow}

Could you generate the sequence of branch conditions encountered on the way to the sink, starting from the source ?

Include *every single* if-else, try-except, or switch statement that the program flow will encounter in the path from the source to the sink.

This should take the form of a sequence of program points, each in the following format:

```
{
  "type": "If-Else | Try-Except | Switch",
  "code": "Source code of program point (1-2 lines)",
  "file": "File path (absolute)",
  "outcome": "What should be the outcome of the branch statement in order to reach the vulnerability?"
}
```

You can use multiple intermediate steps and tool invocations, but when you are finished, your final response should contain the sequence in the above format, within the tags `<SEQUENCE>` and `</SEQUENCE>`.

Listing 4: Branch Reasoning Part 2

`<Conversation continued from Part 1>`

Based on the above branch conditions that you generated, infer a set of conditions that the external input must satisfy in order to reach the vulnerability. Your final answer should be in the following format:

```
<CONDITIONS>
1. Condition 1
2. Condition 2
...
</CONDITIONS>
```

Listing 5: Test Generation

The project I am working with has a vulnerability, reported as a CWE. The issue description says:

```
{description}
You do not have access to the internet or GitHub to look up more details. There are no vulnerability reports in the project directory either.
```

Now create a test case that FAILS (exits with non-zero code) if the vulnerability EXISTS, and PASSES (exits with code 0) if the vulnerability DOES NOT EXIST.

```
{cwe_desc}
This test should actually run the vulnerable code in the project.
```

- It should NOT read the source code to check for the presence of a vulnerability.
- It should NOT `\simulate\` the vulnerability by running some separate code that does not use the project.

Here is a flow consisting of a sequence of program points to reach the vulnerability:

```
{flow}
```

The test should start from the vulnerability 'source' and reach the 'sink'. It should be designed such that it passes through all the branch conditions on the way. This means that the input and method calls should be carefully crafted, satisfying the following conditions:

```
{conditions}
```

The project is built and run as a Docker container, and the Dockerfile is at `{workdir}/Dockerfile.vuln`. All the build dependencies for the project are already installed in `Dockerfile.vuln`. However, if you need any new dependencies, you can add them to `Dockerfile.vuln`.

Make sure to not modify any of the lines in the Dockerfile above `\# Do not modify anything above this line\`. The entire project directory is copied into the Docker container, so you don't need to write any new COPY commands in the Dockerfile. The command to run the test should be the `CMD` command in `Dockerfile.vuln`, so that the test can be run with `docker run -t imagename`.

Feel free to create any new files to create the test case. You are highly encouraged to insert print statements in the existing source files to debug your test. Remember the branch conditions and flow that you derived earlier, and use them to guide your test generation and debugging process.

Once you verify that the flow has reached the 'sink', you should analyze the observed behavior of the program to ensure that the test FAILS if the vulnerability exists, and PASSES if it does not exist. To re-emphasize, this test should NOT be based on reading the source code, but rather on the actual behavior of the program when it is run. If I fix the vulnerability in the project, the test should PASS.

```
{tool_description}
```

If you successfully generate the test case and confirm that it satisfies all the above conditions, respond `<DONE>`.

Listing 6: Repair

The test you generated had the following error:

```
{feedback}
Please fix the test case. Carefully analyze this output for errors or messages that can help you debug your test. Reason step-by-step about what might have gone wrong, and how you can fix it.
```

You can use the `<TOOL>...</TOOL>` format to invoke tools, and you can also add new files.

When you have generated, run and checked your test again, respond with a message containing the string `"<DONE>"`.

Remember that the test should actually run the vulnerable code in the project,

- It should NOT read the source code to check for the presence of a vulnerability.
- It should NOT `\simulate\` the vulnerability by running some separate code that does not use the project.