

Diffploit: Facilitating Cross-Version Exploit Migration for Open Source Library Vulnerabilities

Zirui Chen

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
chenzirui@zju.edu.cn

Xing Hu*

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Zhipeng Xue

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
zhipengxue@zju.edu.cn

Xin Xia*

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
xin.xia@acm.org

Jiayuan Zhou

Queen's University
Kingston, Canada
jiayuan.zhou@queensu.ca

Xiaohu Yang

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
yangxh@zju.edu.cn

Abstract

Exploits are commonly used to demonstrate the presence of library vulnerabilities and validate their impact across different versions. However, their direct application to alternative versions often fails due to breaking changes introduced during evolution. These failures stem from both changes in triggering conditions (e.g., API refactorings) and broken dynamic environments (e.g., build or runtime errors), which are challenging to interpret and adapt manually. Existing techniques primarily focus on code-level trace alignment through fuzzing, which is both time-consuming and insufficient for handling environment-level failures. Moreover, they often fall short when dealing with complicated triggering condition changes across versions. To overcome this, we propose DIFFPLOIT, an iterative, diff-driven exploit migration method structured around two key modules: the Context Module and the Migration Module. The Context Module constructs contexts derived from analyzing behavioral discrepancies between the target and reference versions, which capture the failure symptom and its related diff hunks. Leveraging these contexts, the Migration Module guides an LLM-based adaptation through an iterative feedback loop, balancing exploration of diff candidates and gradual refinement to resolve reproduction failures effectively. We evaluate DIFFPLOIT on a large-scale dataset containing 102 Java CVEs and 689 version-migration tasks across 79 libraries. DIFFPLOIT successfully migrates 84.2% exploits, outperforming the change-aware test repair tool TARGET by 52.0% and the rule-based tool in IDEA by 61.6%. Beyond technical effectiveness, DIFFPLOIT identifies 5 CVE reports with incorrect affected version ranges, three of which have been confirmed. We also discover 111 unreported versions in GitHub Advisory Database.

CCS Concepts

- Security and privacy → Software security engineering.

Keywords

Library Vulnerabilities, Exploit Migration, Affected Version

*Corresponding authors

1 Introduction

Open-source libraries serve as critical infrastructure in modern software development, allowing developers to avoid redundant reimplementation and accelerate the development process [32, 51, 57, 65, 71]. However, the widespread adoption of open-source libraries raises concerns about the risk posed by vulnerabilities in these libraries [10, 23, 37, 42, 43, 45, 64, 68–70], as they can propagate from upstream libraries to downstream projects [5, 33, 35]. Given that libraries often evolve rapidly and downstream projects may depend on a wide range of versions, it becomes essential to assess the impact of vulnerabilities across versions [4, 49]. To this end, developers often employ publicly disclosed exploits to evaluate whether and how vulnerable behavior manifests in different versions [66], such as identifying affected library versions [13, 29] and assessing exploitability in downstream projects with various affected versions [10, 17, 20, 31, 70].

However, simply reusing the original exploit on other affected versions frequently fails to reproduce without adaptation [13, 66], since disclosed exploits are typically crafted for the version in which the vulnerability was originally reported and do not generalize across other affected versions. These failures often result from (1) broken dynamic environments [26, 61, 66] and (2) changes in the underlying triggering condition [67]. Manually understanding and resolving these issues is often time-consuming and requires substantial expertise, highlighting the need for automated exploit migration. Existing studies have demonstrated the feasibility of migrating exploits across versions [13, 29], typically by aligning execution traces via API matching. Despite their success in handling minor API changes, these approaches often rely on fuzzing, which can be time-consuming, especially when multiple versions require migration. More importantly, they overlook two critical challenges that frequently arise in practice:

① **Broken Dynamic Environment:** Previous methods primarily address code-level variations, neglecting the incompatibilities introduced by environmental changes such as dependency upgrades or runtime configurations. For instance, the exploit for CVE-2020-5245 [55] fails with a runtime error in version 1.3.8 due to an updated

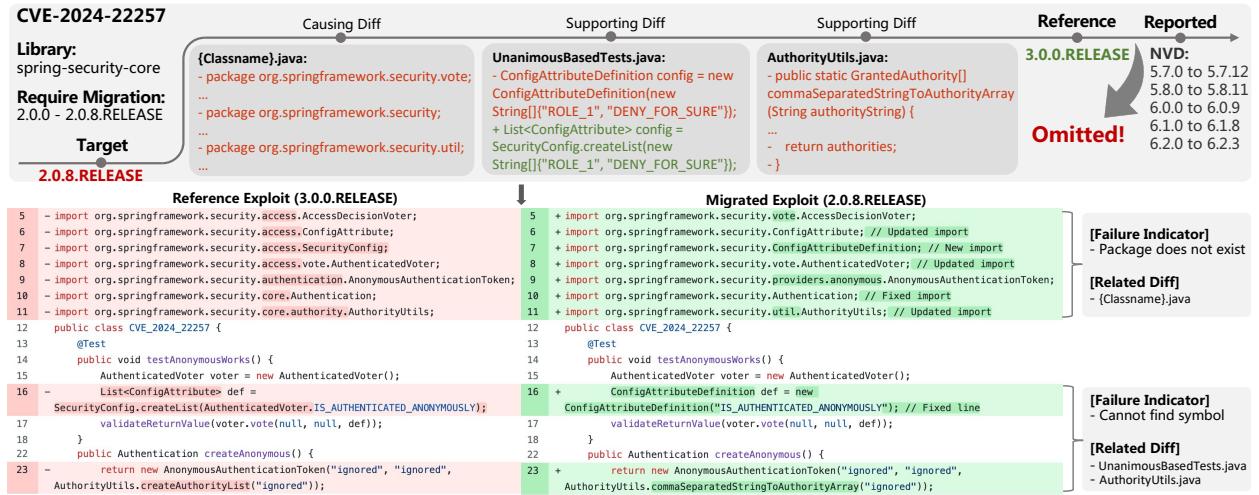


Figure 1: Migrating the exploit of CVE-2024-22257 from version 3.0.0.RELEASE to 2.0.8.RELEASE by DIFFPLOIT.

library (*javassist*), despite identical exploit code. Such environment-induced failures may surface across multiple phases, including the build process and runtime [63], and exhibit considerable variability across libraries, posing significant challenges to mitigation.

② Complicated Triggering Condition Evolution: When API interfaces undergo substantial changes, refactoring [56], or removal between versions, existing trace alignment methods, such as identifying function renaming or function merging/splitting [13], struggle to identify suitable replacements. A representative example is CVE-2024-22257 [16] illustrated in Figure 1, where critical methods (*createList* and *createAuthorityList*) are absent in *2.0.8.RELEASE*, requiring alternative APIs with significantly different signatures and semantics, which cannot be effectively addressed through existing API matching strategies. Incorrect or inadequate matching in these scenarios leads directly to ineffective migration efforts.

Addressing these challenges requires a comprehensive approach that considers both environmental adjustments and intelligent API adaptation mechanisms, beyond simple trace or naming alignment. To effectively address the challenge of exploit migration across versions, we introduce an iterative, diff-driven LLM (Large Language Model) framework, DIFFPLOIT. The key idea is to iteratively adapt failed exploits using version-aware context and feedback. When a migration attempt fails, DIFFPLOIT compares the behavior of target version against a successful reference version to identify failure indicators. These indicators trigger the construction of a structured migration context by the **Context Module**: *Causing Diffs* that likely led to the failure, and *Supporting Diffs* that may help resolve it. This context guides the **Migration Module** to adapt the exploit. A simulated annealing strategy is designed to explore and refine adaptations across multiple iterations. After each attempt, the updated exploit is re-executed, and new feedback is collected to guide the next cycle. Through this closed-loop process, DIFFPLOIT incrementally reduces discrepancies and achieves effective migration, even in the face of significant API or environment changes.

We evaluate DIFFPLOIT on a large-scale open source dataset of Java library vulnerability exploits [57], comprising 102 CVEs and

their explicitly affected versions across 79 libraries. Among them, we identify 30 exploits that require migration across 689 versions, which is comparable to prior studies in the C/C++ domain [13], involving 30 CVEs and 470 versions. Specifically, DIFFPLOIT successfully migrates 580 out of the 689 exploits to target versions in 23 CVEs, achieving a success rate of 84.2%. Compared to the recent change-aware test repair research [48], DIFFPLOIT outperforms TARGET by 52.0% and surpasses the rule-based approach in IDEA by 61.6%, highlighting the effectiveness of DIFFPLOIT in harnessing the capabilities of LLMs for exploit migration. We demonstrate the rationality of DIFFPLOIT's design through ablation studies, where it achieves a 46.1% improvement over the base model. To validate the practicality and correctness of exploits migrated by DIFFPLOIT, we identify previously undisclosed vulnerable versions based on the migrated exploits on five CVE vulnerabilities and contact the CNAs (CVE Numbering Authority) [11] for confirmation. Among them, three cases are confirmed, with our submitted exploit links incorporated into the reference links. Meanwhile, we submit six pull requests to the GitHub Advisory Database to include a total of 111 missing versions detected by DIFFPLOIT, of which 82 versions are accepted. We further demonstrate that DIFFPLOIT effectively overcomes the previously mentioned challenges with a low cost.

This paper makes the following main contributions:

- We propose DIFFPLOIT, a novel discrepancy-driven approach that automatically migrates exploits from target versions to reference versions. Both the source code and dataset of DIFFPLOIT are available on our website [9]
- We conduct an evaluation on 689 versions and demonstrate that DIFFPLOIT successfully migrates exploits for 580 of them, outperforming our baselines.
- Our migrated exploits reveal five CVEs with inaccurate affected version ranges (three confirmed by CNAs) and uncover 111 affected versions missing from the GitHub Advisory Database.

2 Motivation

In this section, we introduce the usage scenario and a motivating example to illustrate the challenges addressed by DIFFPLOIT.

2.1 Usage Scenario

In real-world vulnerability management, our method is designed to support two practical usage scenarios:

(1) Adapting ineffective exploits to confirmed affected versions. Disclosed exploits are crucial for validating whether a specific project is affected by a known upstream vulnerability. However, these exploits are typically crafted for a particular version of a vulnerable library and often fail to function on other affected versions due to code or environment changes. For example, suppose a security advisory confirms that version 4.0.56 of the library *netty-codec-http* is affected by CVE-2021-43797, but the public exploit only works on versions above 4.1.0.CR7 [55]. Developers using version 4.0.56 are left without a working exploit to verify the vulnerability in their context. DIFFPLOIT addresses this gap by automatically adapting existing exploits to these confirmed yet incompatible versions, enabling practical verification in real-world environments.

(2) Discovering previously unreported affected versions. Manually curated vulnerability reports often contain inaccuracies and omissions [7, 12, 18, 30, 41]. When a migrated exploit successfully triggers the vulnerability in a version not listed in public advisories, it suggests that the affected range may be broader than disclosed. Thus, DIFFPLOIT can assist in identifying missing affected versions and improving the completeness of vulnerability databases. It is worth noting, however, that if DIFFPLOIT fails to migrate an exploit to a particular version, this does not necessarily mean the version is unaffected, and additional analysis is required [25], such as introducing commit analysis [2, 8, 60].

2.2 Motivating Example

Figure 1 presents a motivating example from *spring-security-core*, demonstrating how DIFFPLOIT facilitates exploit migration across versions. Through this process, we identify that versions prior to 5.7.0.RELEASE are affected by CVE-2024-22257, although they are not included in the CVE report. For versions above 3.0.0.RELEASE, a publicly disclosed exploit for CVE-2024-22257 is available to reproduce the vulnerability, which serves as the reference exploit. The reference exploit fails to execute directly on earlier versions, such as 2.0.8.RELEASE, because it depends on APIs introduced in later versions (e.g., line 16, line 23) and is affected by breaking changes introduced during updates (refactoring existing APIs to various packages) [52], which requires exploit migration.

Recent advances in artificial intelligence, especially LLMs, demonstrate strong capabilities in various tasks [21, 34, 44, 48, 54, 62, 72], like understanding complex documentation [72] and resolving errors [21, 54]. However, without sufficient contextual information, LLMs often struggle to update exploits using the appropriate APIs of the target versions. Furthermore, in earlier versions, LLMs frequently fail to generate valid API calls due to the lack of corresponding usage patterns in their training data [50]. These limitations hinder the effectiveness of LLMs in generating reliable exploits tailored to target versions. We observe that providing diffs between the target version and the reference versions to LLMs enables them to infer

the root causes of the failures and identify adaptation strategies, such as locating alternative APIs.

Based on this insight, we identify two types of diffs that facilitate exploit migration: *Causing Diff* and *Supporting Diff*. Failures in exploit reproduction, either due to modifications in triggering conditions or environmental breaks, are caused by changes introduced in the evolution of the library (*causing diff*). Meanwhile, libraries themselves often contain internal adaptations to these changes, which can provide LLMs with guidance for migration (*supporting diff*). Take CVE-2024-22257 as an example:

- *Causing Diff*: The cause of the *Package does not exist* error from line 5 to line 11 is the result of refactoring or modifications to the class file. Specifically, the corresponding hunks are those involving target package name changes, such as the removal of the package declaration line *package org.springframework.security* in the *ConfigAttribute* class. These changes provide LLMs with accurate package structure information of the target version, thereby reducing hallucinations such as generating outdated or non-existent import paths.
- *Supporting Diff*: For the modification in line 16, a change in *UnanimousBasedTests* reflects a response to the refactoring of *ConfigAttributeDefinition* and guides the modification in resolving the *Cannot find symbol* error for the *createList* method. In addition, the diff in *AuthorityUtils* provides explicit guidance for replacing the unavailable *createAuthorityList* method at line 23 with an alternative API introduced in the target version. These diffs facilitate LLMs in addressing the evolution of triggering conditions by identifying a suitable replacement for deleted APIs.

Building on the identified diffs, we propose DIFFPLOIT, which iteratively leverages the diffs to facilitate exploit migration through a feedback loop. DIFFPLOIT takes nine steps to migrate the exploit for CVE-2024-22257, despite the significant challenge of identifying relevant information from the 2,555 diffs between the two versions.

3 Proposed Approach

In this section, we present our approach, DIFFPLOIT, detailing the overall workflow as well as the design of its two core components: the **Context Module** and the **Migration Module**.

3.1 Overview

Given a target version requiring exploit migration, our method iteratively repairs its behavioral discrepancies with reference exploits through a self-adaption process. The reference version is chosen from the set of reproduced versions to provide contextual guidance.

The overall process follows a discrepancy-driven loop. At each iteration, we compare the execution outputs of the target version and the reference version to identify reproduction failures. Each detected failure indicator in the target version triggers the construction of a dedicated migration context via the **Context Module**. This context encapsulates the failure symptom, its corresponding diagnostic key, and the associated code-level differences that may either underlie or assist in resolving the failure.

For each detected failure, the corresponding migration context is passed to the **Migration Module**, which attempts to adapt the exploit with the context. The migration process is guided by a simulated annealing strategy that explores diffs within the context

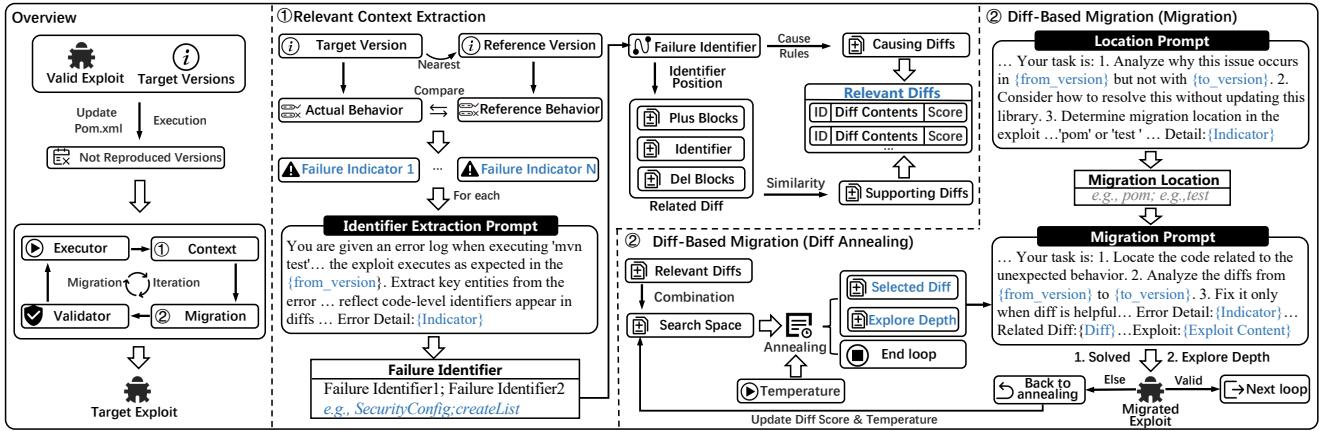


Figure 2: The overall framework and prompt details of DIFFPLOIT.

to identify those effective in resolving the failure. The search process continues until the failure is resolved or a termination criterion (e.g., iteration limit or temperature threshold) is met.

Upon resolving each failure indicator, the adapted exploit is re-executed on the target version to collect fresh output. These outputs are then compared again to extract updated failure indicators. The process iterates through these indicators until all discrepancies are either successfully addressed or deemed irrecoverable under the current search configuration.

3.2 Context Module

The Context Module extracts a set of *migration contexts* from the target version v_t by analyzing behavioral discrepancies with a reference version v_r . Each migration context corresponds to a specific failure unit, representing a reason for reproduction failure (triggering condition change or dynamic environment broken), and includes the related diffs that potentially cause or resolve the failure. These migration contexts provide structured and targeted information to guide the subsequent migration process.

Definition 3.1 (Migration Context). An *migration context* is a structured representation that links a reproduction failure indicator f_i observed in v_t to its corresponding failure identifier k_i , along with related diff hunks. Formally, it is defined as:

$$C_i = (f_i, k_i, D_{\text{cause}}^{(i)}, D_{\text{support}}^{(i)})$$

where:

- $D_{\text{cause}}^{(i)}$ are diff blocks related to the potential root cause of f_i ;
- $D_{\text{support}}^{(i)}$ are diff blocks that provide additional assistance for migration, such as modified test cases or alternative functions.

All diff blocks refer to Git-style diff hunks produced by *git diff*.

3.2.1 Failure Indicator Extraction. For each v_t requiring exploit migration, we determine a corresponding v_r to serve as contextual guidance. The reference version is selected from a list of successfully reproduced versions $R = \{v_1, v_2, \dots, v_n\}$ such that $v_r = \arg \min_{v \in R} \text{dist}(v, v_t)$, where $\text{dist}(\cdot)$ denotes a version distance metric. The list R is constructed by executing the disclosed exploit across all historical versions from the Maven Central repository.

After identifying v_r , we execute the reference exploit against both v_r and v_t , and collect respective outputs. We then compare the two outputs to detect behavioral discrepancies. Each discrepancy is abstracted into a *failure indicator* f_i , representing a concrete symptom of the failure in v_t . Failure indicators encompass a variety of failure types, including build-time errors, runtime exceptions, and assertion failures. The set of extracted indicators $\{f_1, f_2, \dots, f_n\}$ serves as the entry point for constructing migration contexts.

3.2.2 Failure Identifier Extraction. Given a failure indicator f_i , the goal of this module is to derive a corresponding set of failure identifiers k_i that concisely capture the semantic essence of the failure to support downstream diff retrieval. While certain indicators, such as build-time errors, follow structured formats that can be parsed with simple rules, others such as runtime exceptions or unexpected execution outcomes are more variable in syntax and semantics, making them difficult to handle with pattern-based heuristics.

To address this, for each f_i , we prompt the LLM to generate a corresponding set k_i of identifiers, where each element in k_i is a salient token selected to reflect the core failure content in a way that facilitates matching against diff content. These identifiers k_i typically include method names, exception types, or symbolic tokens that are likely to appear in the associated diffs, and will later be used to retrieve causing and supporting diffs.

3.2.3 Causing Diff Extraction. The goal of this module is to identify diff blocks that potentially the root cause of a given failure f_i . We leverage a set of handcrafted heuristic rules tailored to common failure types to extract relevant diffs. These rules allow us to efficiently associate symbolic error traces with corresponding code modifications without relying on complex program analysis.

Given a failure indicator f_i and its corresponding identifier set k_i , we define a mapping function:

$$D_{\text{cause}}^{(i)} = \text{RULE}(\text{type}(f_i), k_i)$$

where $\text{type}(f_i)$ denotes the high-level category of the error (e.g., ‘AssertionMismatch’, ‘MissingMethod’, ‘RuntimeError’). We implement the following rules, which are designed to handle not only triggering condition changes and environment-level breakages but also other unexpected scenarios encountered during migration:

- **AssertionMismatch:** Collect diff blocks that affect method in k_i calls directly invoked in the exploit, especially those on the call stack leading to assertion failure.
- **RuntimeError:** Identify diff blocks that modify methods, classes, or environment associated with the exception type or message tokens found in k_i . These symbols typically reflect failed API usage or behavior changes in dynamic execution.
- **MissingClass / MissingPackage:** Identify diffs under the file corresponding to the missing class or package in k_i .
- **MissingMethod:** Locate diff blocks that modify or remove the method signature referenced in k_i .
- **IncompatibleType / WrongReturn:** Retrieve diff blocks that affect method return types, parameter types, or generic usage near the method identified by k_i .
- **Other:** For all other failure types, we retrieve diff blocks that modify elements (e.g., classes, methods, or fields) whose names match any identifier in k_i . This fallback rule ensures general applicability when more specific heuristics are not available.

We apply a set of rules to determine whether a diff hunk satisfies these conditions (e.g., pattern `\b\w[\w\s>,]*\s+{cause}\s*\(\` is used to detect *MissingMethod* cases). Each diff hunk matched by these heuristics is then assigned a relevance score (10 points for the first five categories and 2 points for each k_i in *Other*) to guide the subsequent migration process.

3.2.4 Supporting Diff Extraction. This module aims to identify diff blocks introduced by the library in response to breaking changes leading to f_i , which can assist the exploit migration process. We observe that breaking changes, such as API deletions or refactorings that cause exploit failures, also affect internal API usages within the library. To maintain functionality and compatibility, the library often introduces corresponding code updates, including modifications to internal method calls, additions of wrapper functions, or adjustments to test cases. These responsive changes serve as valuable supporting for guiding migration.

Such responses frequently appear as newly added diff blocks that are structurally similar to deleted or modified code within the same hunk. To detect such patterns, we first locate occurrences of tokens from the failure identifier set k_i within the diff. Then, around each occurrence, we extract contiguous diff blocks composed of lines sharing the same diff prefix. Specifically, we collect:

- \mathcal{B}_+ : blocks of contiguous added lines (prefixed with '+') near the positions where tokens in k_i appear;
- \mathcal{B}_- : blocks of contiguous deleted lines (prefixed with '-') near the positions where tokens in k_i appear;

We compute token-level similarity between each added block $b_+ \in \mathcal{B}_+$ and corresponding deleted block $b_- \in \mathcal{B}_-$ using the normalized length of their longest common subsequence (LCS):

$$\text{sim}(b_+, b_-) = \frac{2 \times |\text{LCS}(b_+, b_-)|}{|b_+| + |b_-|}$$

where $|\text{LCS}(\cdot)|$ denotes the number of matched tokens in LCS, and $|\cdot|$ denotes the token count of a block. If the similarity exceeds a threshold τ , the diff hunk containing the block pair is considered a candidate hunk and is selected as part of the support diff set $D_{\text{support}}^{(i)}$.

We assign each selected hunk a score based on this similarity (ranging from 0-10), which is then used to guide the subsequent diff annealing process.

Finally, the migration context C_i for failure f_i is generated by associating the failure indicator and its identifier k_i with the extracted causing diffs $D_{\text{cause}}^{(i)}$ and supporting diffs $D_{\text{support}}^{(i)}$, forming a structured representation to guide the migration process.

3.3 Migration Module

To support effective migration, the Context Module provides candidate diff hunks derived from the migration context C_i . These candidates form the search space for the Migration Module and play a central role in navigating towards a successful exploit migration. Building on this set, the Migration Module guides the migration process by selecting diffs with our annealing strategy and presenting them along with f_i to the LLM, which implicitly determines the appropriate migration locations within the exploit.

3.3.1 Diff Annealing. To expand the diversity and complexity of candidate diffs for migration, we design an annealing-based mechanism to explore a broader diff space with controlled randomness. Here, *complexity* refers not only to the number of diffs applied simultaneously to a given failure indicator f_i , but also to the number of repair rounds permitted when new failure indicators emerge during the migration process of f_i . This setting allows the system to consider both wider combinations of patches and deeper chains of repair actions, thereby enriching the potential solution space.

This mechanism operates over three categories of diffs: high-scoring causing diffs D_{cause} , supporting diffs D_{support} , and a newly synthesized set of composite diffs D_{combo} . We begin by constructing D_{combo} through pairwise combinations of top-ranked diffs from $D_{\text{cause}} \cup D_{\text{support}}$. For each combination, the new diff inherits a composite score, computed as the average of its constituent diffs' scores. Together, $D_{\text{cause}} \cup D_{\text{support}} \cup D_{\text{combo}}$ define the search space \mathcal{D} for this annealing process.

To initiate the annealing, we define a temperature parameter T , which governs both the selection probability of diffs and the exploration depth for resolving the current failure indicator f_i . The selection probability of a diff $d \in \mathcal{D}$ is proportional to its normalized score and temperature:

$$P(d) \propto \exp\left(\frac{s(d)}{T}\right),$$

where $s(d)$ denotes the score of diff d . Higher temperatures bias the selection toward higher-scoring diffs.

In addition, the *exploration depth*—i.e., the number of retry attempts made when new failure indicators emerge during migration—is inversely correlated with the temperature. At higher temperatures, the system prefers quick evaluations of promising diffs, while deeper exploration is deferred to later stages when the temperature cools down. This design ensures that the system prioritizes efficient evaluation of high-scoring diffs in early stages.

During each iteration:

- A diff $d \in \mathcal{D}$ is sampled and applied to perform a migration targeting f_i via LLM.

- If the migration fails, the score of d is penalized, the global temperature T is decreased, and the diff combination list D_{combo} is refreshed based on the updated scores.
- A new diff is then selected, and the process repeats.

This annealing-based strategy enables a controlled exploration of the diff space, balancing *exploitation* of high-scoring diffs and *exploration* of more diverse or complex diff combinations.

3.3.2 Exploit Migration. Given a failure indicator f_i extracted from the execution outputs of the target version, the Exploit Migration module aims to adapt the original exploit such that the expected vulnerability behavior is restored. This process takes as input a selected related diff d_i , which is expected to resolve the failure based on prior analysis.

We first utilize the LLM to localize the relevant modification site within the exploit, conditioned on the failure indicator f_i , its associated diagnostic key k_i , and the selected diff d_i . These elements are encoded into a structured prompt, as illustrated in our Figure 2, which guides the LLM in pinpointing the modification location.

Based on this localization, DIFFPLOIT then instructs the LLM to generate an adapted exploit \hat{e}_i by modifying the original exploit according to the changes reflected in d_i at the identified position. \hat{e}_i is then executed on the target version. Its runtime output is analyzed to determine whether f_i has been resolved.

3.3.3 Migration Validation. Following the adaptation of the exploit targeting the specific failure indicator f_i in the preceding module, DIFFPLOIT re-run the exploit to collect the updated failure indicators in order to determine whether f_i has been resolved.

- If f_i still appears in the failure list, the migration attempt is considered unsuccessful. The process then returns to the annealing process to select the next diff candidate. This iterative search continues until f_i is resolved or the search terminates due to reaching the temperature limit or timeout.
- If f_i no longer appears in the failure list, indicating successful resolution, we analyze the updated failure indicators further:
 - If no failure indicators remain, the migration process proceeds to perform the final reproduction verification by checking whether the assertions in the exploit behave as expected and whether the observed behavior matches the expected reproduction behavior. If both conditions are satisfied, the migration process terminates successfully.
 - If the remaining failure indicators are all previously known (i.e., identified prior to addressing f_i), we consider f_i to have been successfully resolved and proceed to handle the remaining failures accordingly.
 - If new failure indicators emerge after attempting to resolve f_i , we initiate a limited number of additional repair attempts to address these emergent failures. The number of such attempts, denoted as the *exploration depth*, is determined based on the current temperature T , with higher annealing temperatures yielding shallower exploration. If the number of repair attempts exceeds the exploration budget derived from T , the current migration path is terminated, and a new diff candidate is selected to resolve f_i .

4 Experimental Setup

Research Questions. Our experimental evaluation aims to answer the following research questions:

- **RQ1 (Effectiveness):** To what extent can DIFFPLOIT effectively migrate exploits to target versions?
- **RQ2 (Ablation Study):** How does each component within DIFFPLOIT contribute to the overall migration process?
- **RQ3 (Practical Feasibility):** Is DIFFPLOIT acceptable in real-world scenarios, considering the cost and quality of exploits?

We address RQ1 to evaluate the effectiveness of our diff-based exploit migration method and its superiority over existing approaches. We address RQ2 to assess the rationality of key components in our method, including the causing diff extraction module, the supporting diff extraction module, and our proposed diff annealing algorithm. We address RQ3 to verify whether the exploits generated by DIFFPLOIT and the under-reported vulnerable versions it identifies are recognized and accepted.

4.1 Dataset

To minimize bias in the exploit collection process, we evaluate the performance of DIFFPLOIT using the largest publicly available Java exploit dataset [57]. This dataset comprises 102 CVE vulnerabilities, each with a corresponding exploit targeting a specific version and a set of manually verified affected versions, making it well-suited for assessing the effectiveness of exploit migration approaches.

To identify versions requiring exploit migration, we execute exploits on versions labeled as affected and examine execution results. The dataset includes assertions designed to verify reproduction, which means versions failing to satisfy assertions are marked as requiring migration. We identify 988 versions meeting this criterion. Further analysis is performed to confirm the presence of vulnerabilities in these versions, resulting in a final set of 689 truly affected versions for 30 vulnerabilities. This reduction is mainly attributed to 176 versions in CVE-2023-51080 and 72 versions in CVE-2021-43795 that are incorrectly labeled as vulnerable before the introduction of the vulnerability [38, 53].

4.2 Baselines

To the best of our knowledge, no prior work has explored exploit migration across Java library versions, though third-party libraries play a crucial role in the Java ecosystem [22]. Existing studies [13, 29] rely on fuzzing frameworks such as AFLGo [6], which are difficult to adapt to Java. We include the following four baselines: ① TARGET [48] is a pre-trained language model-based approach for automated function-level test repair, which treats test repair as a language translation task and leverages context information extracted from the test breakage. Although it is not specifically designed for exploit migration, we include TARGET as a baseline due to its strong performance in repairing broken JUnit test cases, which are structurally similar to the exploits in our dataset. ② IDEA [27] is the combination of Quick Fix and Auto-import features provided by IntelliJ IDEA. These features assist developers in resolving compilation issues caused by API refactoring, missing imports, or outdated method signatures, which can partially mitigate exploit failures caused by changes in triggering conditions. ③ GPT-4o and ④ DeepSeek-v3 are two of the most advanced proprietary LLMs,

demonstrating strong capabilities in both code understanding and generation. In this study, we examine whether existing SOTA LLMs can generalize to the task without any task-specific fine-tuning.

4.3 Migration Success Criteria

To ensure an exploit triggers the same vulnerability after migration, we perform validation along two dimensions: assertion consistency and behavioral verification. Assertion consistency refers to whether the migrated exploit exhibits the same assertion failure behavior as the original, indicating consistency at the assertion level. Behavioral verification involves checking fine-grained behavioral indicators in the output logs to determine whether the expected vulnerability behavior is preserved after migration.

An exploit is considered successfully migrated to the target version if and only if it triggers the expected assertion, and the location and manifestation of the assertion are consistent with those in the reference version.

4.4 Implementation

In our experimental setup, we deploy a Docker environment based on Ubuntu 20.04. Following the configuration by Wu et al. [57] for exploit collection, we select Java 11 as the runtime environment, specifically version 18.9 (version: build 11+28). We execute exploits using the `mvn test` command. For the selection of LLMs, we employ a high-performance closed-source model, GPT-4o (snapshot as of 2024-11-20), alongside an open-source model with strong performance on code tasks, DeepSeek-v3 (snapshot 0324), to conduct our experiments. To improve reproducibility, the base model of DIFFPLOIT is DeepSeek-v3. A time constraint of 5 minutes is applied to DIFFPLOIT and baselines for each version during experiments. Since all libraries in our dataset are hosted on GitHub/GitLab, we use `git diff` to generate the diff files.

Regarding the baseline setup, we reproduce TARGET using the fine-tuned weight provided in the replication package. Despite our best efforts, we were unable to reproduce TARGET on CVE-2020-13956 and CVE-2023-51075, as it requires the construction of valid exploits tailored to these vulnerabilities. For experiments involving IntelliJ IDEA, we use version 2025.1.3. We apply IntelliJ IDEA Quick Fix and Auto-import features to modify the exploit and run tests using `mvn test` after no further modifications are possible. When suggestions are abundant, including cases like multiple renaming suggestions, we evaluate the top five candidates.

5 Experimental Evaluation

We evaluate the performance of DIFFPLOIT from three perspectives. First, we assess its effectiveness and compare it with baseline methods using the largest dataset of Java library vulnerability exploits, and analyze its strengths and limitations. Second, we conduct an ablation study to demonstrate the contributions of the diff and migration modules to the overall performance. Finally, we analyze the practical feasibility of DIFFPLOIT based on cost and responses from CNAs and open-source maintainers.

5.1 Effectiveness

5.1.1 Performance. We evaluate the effectiveness of DIFFPLOIT on a dataset containing 689 version pairs that require exploit migration. DIFFPLOIT successfully migrates 580 of them, achieving an overall success rate of 84.2%. In the context of 30 representative CVEs, DIFFPLOIT successfully performs exploit migration for 23 cases, covering a diverse range of vulnerability types and affected libraries. This demonstrates the generality of DIFFPLOIT across both vulnerability classes and dependency ecosystems. Notably, for 20 CVEs, all associated versions requiring migration are successfully repaired, highlighting the potential of DIFFPLOIT to serve as a reliable component in automated vulnerability validation pipelines. However, it is important to emphasize that a failed migration does not necessarily imply the absence of a vulnerability.

Compared to the baseline method TARGET, DIFFPLOIT achieves a relative improvement of 51.0%, demonstrating its robustness in handling critical challenges in exploit migration, such as adapting to non-function-level edits like import adjustments and build configuration updates. When compared with IDEA that combines IntelliJ IDEA’s Quick Fix and Auto Import features, DIFFPLOIT outperforms significantly in 61.6%. This result highlights its ability to address migration scenarios that exceed the capabilities of predefined rules. We also compare DIFFPLOIT against direct application of LLMs. GPT-4o and DeepSeek-V3 can migrate 243 (35.2%) and 263 (38.1%) cases respectively, which, while competitive in some straightforward scenarios, underperform due to lack of migration context. In contrast, DIFFPLOIT leverages migration-specific contextual information, enabling it to maintain a higher success rate especially in complex migration scenarios.

5.1.2 Strength. Compared to the baseline method TARGET, DIFFPLOIT demonstrates superior adaptability in handling test changes that are not confined to the function level. While TARGET focuses on identifying faulty functions, it relies on developers to manually apply legitimate edits when changes occur outside function bodies, such as modifications in import statements or resolving runtime environment broken in pom.xml. As a result, it fails to repair test cases in 16 CVEs where such non-functional changes are essential.

DIFFPLOIT is also capable of addressing scenarios where an exploit test should fail but instead passes silently in the target version. These cases, where the presence of a vulnerability is masked by a superficially successful test, are often overlooked by TARGET. By leveraging rich migration context and historical diffs, DIFFPLOIT can detect and adapt such misleading test cases, ensuring they remain effective indicators of vulnerabilities.

Additionally, DIFFPLOIT leverages a structured migration context that captures both the causing and supporting diffs of a test case. This design enables it to accommodate a broader range of code modifications related to exploit migration. This advantage becomes more prominent in complex exploit migration tasks, which require understanding subtle semantic changes. In our Discussion section, we further analyze the edit distance before and after migration, highlighting DIFFPLOIT’s ability to apply complex adaptations.

5.1.3 Limitations. While DIFFPLOIT demonstrates strong performance in migrating exploits across versions, it has several limitations. First, it struggles to handle cases where the vulnerability

Table 1: Performance of DIFFPLOIT and Baseline Methods on Exploit Migration

Library	CVE	Reference Version	Affected Versions		DIFFPLOIT	TARGET	IDEA	LLM	
			Total	Need Mig.				GPT-4o	DeepSeek-v3
commons-fileupload	CVE-2016-1000031	1.1.1	8	3	3	0	0	3	3
cfx-rt-rs-security-xml	CVE-2014-3584	2.6.10	33	14	0	0	0	0	0
dolphinscheduler-api	CVE-2022-34662	2.0.0	20	12	0	0	0	0	0
dolphinscheduler-common	CVE-2023-49250	2.0.0-alpha	44	12	0	0	0	0	0
dropwizard-validation	CVE-2020-5245	2.0.1	40	16	16	0	0	0	0
hibernate-validator	CVE-2019-10219	6.0.5.Final	31	12	12	0	0	1	12
httpclient	CVE-2020-13956	4.5.3	40	6	0	–	0	0	0
hutool-core	CVE-2023-51075	5.7.18	38	1	0	–	0	0	0
jackson-databind	CVE-2022-42004	2.0.0-RC1	24	5	2	0	0	0	0
jackson-dataformat-xml	CVE-2016-7051	2.7.7	75	69	69	0	0	0	0
junrar	CVE-2022-23596	6.0.0	15	8	8	0	0	5	8
kernel	CVE-2022-24197	7.2.0	24	5	0	0	0	0	5
		CVE-2021-43797	4.1.49.Final	157	86	86	80	0	80
netty-codec-http	CVE-2019-20444	4.1.43.Final	131	71	71	65	65	1	65
		CVE-2019-16869	4.1.0.Beta1	131	71	71	65	65	1
netty	CVE-2015-2156	3.3.0.Final	65	45	0	0	0	0	0
para-core	CVE-2022-1848	1.42.2	102	84	84	0	21	84	23
plexus-utils	CVE-2017-1000487	1.4.2	50	2	2	2	0	2	2
postgresql	CVE-2024-1597	9.4.1212	179	49	48	0	0	0	0
protocols-imap	CVE-2021-40111	3.5.0	11	9	9*	0	0	0	0
socket.io-client	CVE-2022-25867	1.0.0	12	9	9	0	0	0	0
spring-amqp	CVE-2017-8045	2.1.0.RELEASE	49	2	2	0	0	2	2
spring-actuator-logview	CVE-2021-21234	0.2.9	14	10	10	10	5	5	10
spring-context	CVE-2022-22968	4.2.9.RELEASE	194	19	19	0	0	19	19
spring-security-core	CVE-2019-11272	3.0.0.RELEASE	61	9	9	0	0	0	0
		CVE-2024-22257	5.7.11	210	9	9	0	0	9
spring-webmvc	CVE-2014-3625	3.0.4.RELEASE	30	1	1	0	0	1	1
spring-web	CVE-2013-6430	1.1.1	53	4	4	0	0	0	0
wicket-core	CVE-2013-2055	1.5.10	33	7	7	0	0	0	0
SUM	30 CVEs	–	1,998	689	580 (84.2%)	222 (32.2%)	156 (22.6%)	243 (35.2%)	263 (38.1%)

*The exploit is a flaky test.

manifests differently across versions. For instance, in CVE-2020-13956 and CVE-2023-51075, the exploits trigger exceptions such as *NumberFormatException* and *IndexOutOfBoundsException*, rather than the originally expected assertion failures. Although such runtime exceptions still indicate security-relevant behavior, DIFFPLOIT currently treats them as failures, as it enforces a fixed assertion-based validation strategy. Similarly, in CVE-2023-49250, the exploit leads to a malicious server connection.

```

ResourcesServiceImpl res = new ResourcesServiceImpl();
Method method_verifyFile = res.getClass().getDeclaredMethod("verifyFile", ...)

✓ Migration Success ✘ Diffploit ✘ Migration Failed ✘

ResourcesService res = new ResourcesService();
Method method_verifyFile = res.getClass().getDeclaredMethod("updateResource", ...)

✓ Valid v1.3.9 ✘ Reference v2.0.0-alpha
  
```

Figure 3: A failure case of DIFFPLOIT for CVE-2022-34662.

Second, DIFFPLOIT encounters difficulties when migrating exploits that depend on version-specific APIs, especially when the target version lacks both a structurally similar counterpart and supporting information. As illustrated in Figure 3, in CVE-2022-34662, the original exploit in a higher version invokes *ResourcesServiceImpl.verifyFile*, while the lower version provides a semantically

related method *updateResource* under a different class, *ResourcesService*. DIFFPLOIT failed to migrate the method due to missing contextual alignment. This example shows that while DIFFPLOIT can partially adapt such cases, external API knowledge is required.

5.2 Ablation Study

Our ablation study aims to achieve two goals: (1) to demonstrate that each component in our design contributes to higher exploit reproduction success, and (2) to show that our design helps reduce the reproduction cost in terms of step count. We construct three ablated variants of DIFFPLOIT to evaluate the contribution of each component: (a) **DIFFPLOIT-Causing**, which disables the extraction of causing diffs; (b) **DIFFPLOIT-Supporting**, which disables the extraction of supporting diffs; and (c) **DIFFPLOIT-Annealing**, which removes the diff annealing process. To further evaluate the effectiveness of our diff combination strategy, we design an additional variant named (d) **DIFFPLOIT-Combining**, in which only diff scores are used to provide contextual information, without incorporating among diffs. The performance of the base model without any diff information has already been evaluated in Table 1, so we do not include a separate variant for this setting.

We evaluate each variant using three metrics reported in Table 2. **Average Step** measures the average number of adaptation steps required to successfully migrate exploits, where failures are

assigned a default value of 30 steps, derived from the estimated time to complete exploit execution and the response latency of the LLM within five minutes. **Success Rate** reports the version number and percentage of exploits successfully migrated. **Overhead** quantifies the average step overhead relative to the origin DIFFPLOIT, offering a normalized view of the cost efficiency.

Table 2: Ablation Study on DIFFPLOIT.

Method	Average Step*	Success Rate	Overhead
DIFFPLOIT	8.28	580/689 (84.2%)	-
DIFFPLOIT-Causing	12.71	470/689 (68.2%)	277.15%
DIFFPLOIT-Supporting	11.15	489/689 (71.0%)	159.09%
DIFFPLOIT-Annealing	15.02	462/689 (67.1%)	444.69%
DIFFPLOIT-Combining	13.35	505/689 (73.3%)	400.92%

* Failures are assigned a default number of steps.

We observe from Table 2 that DIFFPLOIT achieves the highest success rate, outperforming the best ablated variant by over 10.9%. This confirms that each component in our design contributes to the overall success. Among the variants, although the success rates degrade when removing any single module, they all remain notably higher than the base model (evaluated separately in Table 1), indicating that the use of context-aware diffs and annealing strategies brings substantial benefits. In terms of cost, DIFFPLOIT not only achieves the highest success rate but also requires the fewest steps on average (8.28), demonstrating that each component of DIFFPLOIT contributes effectively to the overall performance. Although all three variants yield similar success rates, they incur different levels of reproduction cost, with the variant without *Annealing* performing the worst (15.02 steps on average, 444.69% overhead relative to the origin DIFFPLOIT). This highlights the importance of the annealing process in filtering and prioritizing diffs that are most helpful to migration. Although DIFFPLOIT-Combining achieves the highest success rate among all variants, it requires significantly more steps than DIFFPLOIT, indicating that our combination strategy substantially improves the efficiency of discovering valid exploits.

5.3 Practical Feasibility

5.3.1 Acceptability. We aim to introduce an objective assessment of DIFFPLOIT’s practical feasibility by assessing whether migrated exploits are accepted by real-world vulnerability management processes. Some adapted exploits target early branches that are no longer maintained, making it impossible to evaluate the acceptance of DIFFPLOIT by library maintainers. To overcome this, we evaluate the acceptance of DIFFPLOIT by submitting the previously undocumented affected versions supported by our migrated exploits. We identify and report previously undocumented affected versions to CVE, the most authoritative vulnerability repository, and the GitHub Advisory Database, which facilitates communication through pull request-based submissions. Our migrated exploits serve as supporting evidence in these submissions. We assess the real-world acceptance of DIFFPLOIT based on the responses from CNAs and open-source reviewers.

DIFFPLOIT successfully migrates exploits to 580 affected versions. We further investigate whether these versions are explicitly documented in the NVD descriptions and the affected product listings

in the GitHub Advisory Database. As illustrated in Table 3, our migrated exploits uncover five NVD entries with missing or unclear specifications of affected version ranges, as well as 111 affected versions omitted from the GitHub Advisory Database. We contact the corresponding CNAs via email, and three of them have incorporated our migrated exploit into the reference links of the CVE report, while the remaining CNAs have not responded by the time of submission. We also submit six pull requests to the GitHub Advisory Database, which result in updates to 57 affected versions. The remaining submissions are not merged due to GitHub’s limited capacity to validate exploits at scale [3]. The addition of our exploits to the CVE reference links and the update of affected versions identified by migrated exploits demonstrate our practical feasibility.

Table 3: Real World Response of DIFFPLOIT.

CNA	CVE	CNA Response	GitHub Missing
Mitre	CVE-2019-20444	Confirmed	14 (12 Accepted)
Mitre	CVE-2019-16869	Confirmed	2 (0 Accepted)
GitHub, Inc.	CVE-2021-43797	Under Review	13 (Under Review)
GitHub, Inc.	CVE-2020-5245	-	19 (7 Accepted)
VMWare	CVE-2024-22257	Under Review	-
Red Hat, Inc.	CVE-2013-6430	-	38 (38 Accepted)
Red Hat, Inc.	CVE-2019-10219	Confirmed	25 (25 Accepted)
SUM	7 CVEs	3 Confirmed	111 (82 Accepted)

DIFFPLOIT contributed to improving the quality of the CVE reports by identifying additional affected versions and providing a working exploit. As confirmed by Red Hat: “*Thank you again for helping us improve the CVE records. The changes for version information were made, and the references you sent added.*”

5.3.2 Cost. We estimate the financial cost of using DIFFPLOIT to migrate an exploit with base model DeepSeek-V3. On average, each successfully migrated exploit consumes 7,029 input tokens and 831 output tokens. According to the pricing details provided by DeepSeek as of July 2025, this corresponds to an average cost of \$0.0014 per successful migration. Similarly, for failed migrations, the average token consumption is 12,977 input tokens and 997 output tokens, resulting in an average cost of \$0.0020 per failed attempt. These results indicate that the financial cost of using DIFFPLOIT remains low and practical for real-world deployment.

5.3.3 Data Leakage. To evaluate the performance of DIFFPLOIT on unseen vulnerabilities, we conduct an experiment using exploits that are disclosed after the model cutoff date. Our base model used for all experiments is *Deepseek-V3-0324*, thus we select five CVEs that are published after March 2025.

As illustrated in Table 4, DIFFPLOIT successfully migrates exploits for all affected versions of four unseen CVEs, with only two versions of CVE-2025-53864 failing to produce valid exploits. These results demonstrate the robustness of our approach when handling vulnerabilities that the underlying LLM has never seen before.

6 Discussion

This section first examines the extent to which DIFFPLOIT addresses the identified challenges. We then discuss potential threats to validity and how they might affect the interpretation of our results.

Table 4: Evaluation on Vulnerabilities Disclosed Post Cutoff.

CVE-ID	Published Date	Affected Versions		Success
		Total	Need Mig.	
CVE-2025-59340	2025-09-17	92	19	19
CVE-2025-58056	2025-09-03	227	71	71
CVE-2025-53864	2025-07-10	268	192	190
CVE-2025-52999	2025-06-25	183	28	28
CVE-2025-48976	2025-06-16	12	2	2

6.1 Challenge Resolution

We further discuss how DIFFPLOIT addresses the challenges we proposed. We examine the effectiveness of DIFFPLOIT in addressing two primary challenges: ① failures caused by environment broken, and ② the complexity of exploit migration process.

① Broken Dynamic Environments. To understand whether DIFFPLOIT addresses failures caused by broken dynamic environments, we examined the modification locations in migrated exploits. Among the 23 exploits that are successfully migrated, seven of them do not require modifications to the exploit code. These migrations enabled DIFFPLOIT to successfully reproduce 128 previously failing versions, suggesting that many failures of origin exploits are not rooted in the exploit logic but rather caused by external or environmental factors. We further categorized the types of failures that DIFFPLOIT handled without modifying the exploit code. These include (1) incompatibilities caused by other dependencies required by the exploit, such as mismatched versions of *spring-web* and *spring-test*; (2) build-time issues caused by blocked or obsolete repositories in the Maven configuration, such as *repo.spring.io*; (3) runtime failures due to Java platform evolution (e.g., JAXB in Java 11); and (4) dependency resolution failures related to outdated or snapshot artifacts (e.g., *commons-io*, *javax.faces:jsf-api*), or conflicts in third-party libraries (e.g., *Javassist*). These results demonstrate that DIFFPLOIT effectively addresses failures stemming from environmental issues by reconstructing the necessary build and runtime context in the pom file.

② Complicated Migration. We assess the migration complexity of DIFFPLOIT using the Average Edit Distance (AED), a widely adopted metric for migration tasks. AED is defined as the average token-level Levenshtein distance computed over a set of exploit pairs. Specifically, we calculate the AED between exploits before and after migration to evaluate the overall complexity of exploits migrated by DIFFPLOIT. Additionally, to estimate the manual effort required when relying solely on IDEA for migration, we compute the AED between the exploits modified using IDEA alone and the corresponding valid exploits produced by DIFFPLOIT. This comparison quantifies the reduction in manual effort achieved by DIFFPLOIT.

On the 22 CVEs where exploit migration is successful by DIFFPLOIT, the average edit distance between the migrated exploits and the original exploits is 176.00, indicating that the required modifications are non-trivial and reflect substantial structural or semantic changes. In contrast, IDEA is able to provide useful suggestions for only 7 of these cases. Even after applying its suggestions, the resulting exploits still require an average edit distance of 59.69 to match valid exploits produced by DIFFPLOIT. Based on these observations,

we estimate that DIFFPLOIT reduces manual effort by approximately 80%, as measured by the total edit distance required for the exploit migration compared to IDEA. This highlights DIFFPLOIT’s ability to both handle complex migration scenarios and significantly alleviate the manual burden associated with exploit migration.

6.2 Qualitative Analysis

As discussed in Section 5.1.3, we have analyzed the limitations of our approach. To complement that discussion, we further conduct a qualitative analysis of failure cases, which can be broadly categorized into three root causes. Understanding these categories helps explain the remaining gaps and guide future improvement.

Assertion mismatch. Diffploit enforces the original assertions during migration. Some vulnerabilities exhibit different behaviors on different versions, leading to the original exploit assertion being ineffective on the target versions. Table 5 lists representative examples where the observed behavior diverged from the origin.

Table 5: Examples of Assertion Mismatch.

CVE	Expected	Actual
CVE-2020-13956	Wrong Return Value	NumberFormatException
CVE-2023-51075	Timeout	IndexOutOfBoundsException
CVE-2023-49250	Console Output	Remote Server Connection
CVE-2022-42004	StackOverflowError	Unexpected Exception
CVE-2024-1597	Wrong Return Value	VM crash

Domain-specific inputs. Some exploits require highly specialized, format-specific payloads that cannot be synthesized reliably from code diffs or shallow context. Examples include crafted PDFs (e.g., CVE-2022-24197) and structured tokens or cookie names (e.g., CVE-2014-3584, CVE-2015-2156). These artifacts often need precise byte-level or format-aware modification, which is outside the current capabilities of DIFFPLOIT.

Large API gaps. When the target version introduces substantial structural changes (multiple co-occurring edits, API renames, or replaced components), the search and edit synthesis of DIFFPLOIT may only partially complete the necessary transformations. For example, CVE-2022-34662 required replacing a service class and finding an appropriate replacement API (from *ResourcesServiceImpl.verifyFile* to *ResourcesService.updateResource*); Diffploit is only able to perform part of the migration, leaving the exploit nonfunctional.

6.3 Threats to Validity

Our study mainly suffers from the following threats to validity:

External Validity The major threat lies in the generality of the dataset. To overcome this, we evaluate DIFFPLOIT using the largest publicly available Java exploit dataset. However, we acknowledge that the performance on other programming languages remains unverified. Another threat is that the affected version range is manually labeled, which may introduce human errors. Although we perform a second-round validation and eliminate 22.7% of incorrect labeling, there may still be misclassified vulnerable versions. DIFFPLOIT relies on source code to obtain diffs between library versions through the corresponding GitHub repository. However, for some projects, version diffs are no longer publicly available,

such as versions of *spring-framework* prior to 3.0.0. In such cases, DIFFPLOIT may fail due to the lack of accessible diffs.

Internal Validity. One potential threat to internal validity is that the performance of DIFFPLOIT may result from the base model’s prior exposure to migrated exploits, rather than its true generalization capability. To examine this, we conduct an experiment where only the base model is used to perform exploit migration. We observe that the success rate drops significantly when the full method is replaced with the base model alone. This result suggests that the effectiveness of DIFFPLOIT does not stem from potential data leakage but from its diff-based design.

Another threat to internal validity lies in DIFFPLOIT depends on the mvn command for building ('mvn compile') and managing library versions ('mvn versions:use-dep-version'), limiting its applicability to other programming languages. Further modifications to the compilation and exploitation process are required. DIFFPLOIT relies on 'git diff' to generate the diff file between the target and reference versions, which limits its applicability to other version control systems.

7 Related Work

Our work focuses on the task of **exploit migration**, which involves adapting existing exploits to different software versions. This task is crucial for accurate vulnerability assessment and exploit reproduction in real-world scenarios, where environmental differences and software evolution often break naive exploits. Recent studies have proposed various approaches to tackle exploit migration and cross-version exploitability assessment: AEM [29] introduces an automated exploit migration method targeting Linux kernels, which aligns execution points in different kernel versions to reproduce exploitation behaviors. Similarly, VulScope [13] leverages directed fuzzing to migrate exploits between software versions, improving vulnerability detection coverage. SyzBridge [73] addresses environmental differences between upstream Linux kernels and downstream distributions by adapting exploits accordingly, enhancing exploitability assessment accuracy. Evocatio [28] automatically generates exploits to expose previously unknown bug capabilities. These efforts underscore the complexity and practical importance of exploit migration. The aforementioned approaches rely on fuzzing and require explicit execution trace mapping to support exploit migration, which is time-consuming and challenging.

Despite its importance in vulnerability analysis, exploit migration remains underexplored in Java vulnerabilities, motivating our approach, which handles both triggering condition change and environment breakage in a diff-driven framework.

We observe that exploit migration shares similarities with **test migration** and **API migration**. Like automated test repair [1, 14, 47], automated test migration has attracted increasing attention as software systems evolve rapidly, requiring frequent updates to maintain test suite reliability and effectiveness [15, 40]. Prior work includes TARGET, which leverages pre-trained language models to treat test repair as a language translation task [24, 48]. UTFix focused on repairing python unit tests affected by changes in focal methods, using contextual static and dynamic code slices and failure messages [46]. Earlier frameworks such as TestCareAssistant repaired or generated tests by adapting to limited changes

like method parameter additions [39], while TestFix uses genetic algorithms to synthesize method call insertions and deletions to fix broken JUnit tests, albeit only supporting single-assertion tests [59]. To better preserve test intent, TRIP employs a search-based approach guided by dynamic symbolic execution to prioritize repair candidates that maintain original test semantics, generating fixes for tested cases [36]. Similar to test migration, API migration aims at updating API usages in downstream projects to resolve library updates [19, 58, 67]. These approaches can partially address exploit failures caused by API changes during library evolution. Despite these advances, test migration and API migration remain challenging due to the diversity of exploit ineffective reasons. Moreover, existing tasks typically focus only on function-level test repair, overlooking the challenges posed by environmental factors. Addressing such environmental challenges remains an underexplored area.

8 Conclusion

In this work, we present DIFFPLOIT, a novel framework for exploit migration that combines LLM-driven adaptation with dynamic context construction based on version-specific code diffs. By identifying both causing and supporting diffs and employing a simulated annealing strategy to guide iterative adaptation, DIFFPLOIT transforms failure symptoms into actionable migration steps. This feedback-driven process enables reliable exploit transfer across complex version gaps. Through extensive experiments on the largest Java vulnerability exploit dataset to date, DIFFPLOIT achieves a high success rate of 84.2%, outperforming state-of-the-art baselines by a significant margin. Our ablation study confirms that each design component contributes meaningfully to both effectiveness and cost-efficiency. Moreover, real-world responses from CNAs and open-source maintainers affirm the practical feasibility of our approach, with 82 affected versions accepted into GitHub Advisory Database and CNAs of three CVEs confirm affected versions identified based on our migrated exploits. While DIFFPLOIT demonstrates strong adaptability and generality, it remains limited in handling modified reproduction behaviors. These challenges point to promising future directions. Overall, our study shows that DIFFPLOIT not only enhances exploit usability across versions but also contributes to the completeness and accuracy of vulnerability databases.

Acknowledgement

This research is supported by the National Key R&D Program of China (No.2024YFB4506400). We also thank the anonymous reviewers for their insightful comments and suggestions.

References

- [1] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (Porto de Galinhas, Brazil) (FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839
- [2] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding Bug Inducing Commits from Failures. In *Proceedings of the 45th International Conference on Software Engineering (Melbourne, Victoria, Australia) (ICSE '23)*. IEEE Press, 589–601. doi:10.1109/ICSE48619.2023.00059
- [3] Anonymous. [n. d.]. *Pull request for updating CVE-2019-16868*. <https://github.com/github/advisory-database/pull/5774>

- [4] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) (ICSE '22). Association for Computing Machinery, New York, NY, USA, 2352–2364. doi:10.1145/3510003.3510113
- [5] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20 (2015), 1275–1317.
- [6] Marcel Böhme, Van-Thanh Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) (CCS '17). Association for Computing Machinery, New York, NY, USA, 2329–2344. doi:10.1145/3133956.3134020
- [7] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) (ESEC/FSE 2017). Association for Computing Machinery, New York, NY, USA, 396–407. doi:10.1145/3106237.3106285
- [8] Xiao Chen, Hengcheng Zhu, Jialun Cao, Ming Wen, and Shing-Chi Cheung. 2025. SemBIC: Semantic-Aware Identification of Bug-Inducing Commits. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE062 (June 2025), 23 pages. doi:10.1145/3715781
- [9] Zirui Chen. [n. d.]. *Replication Package*. <https://github.com/chen-zirui/PoCAdaptation>
- [10] Zirui Chen, Xing Hu, Xin Xia, Yi Gao, Tongtong Xu, David Lo, and Xiaohu Yang. 2024. Exploiting Library Vulnerability via Migration Based Automating Test Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 228, 12 pages. doi:10.1145/3597503.3639583
- [11] The MITRE Corporation. [n. d.]. *CNA Detail*. <https://www.cve.org/PartnerInformation/ListOfPartners>
- [12] Roland Croft, M. Ali Babar, and M. Mehdi Khloosi. 2023. Data Quality for Software Vulnerability Datasets. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, 121–133. doi:10.1109/ICSE48619.2023.00022
- [13] Jiarun Dai, Yuan Zhang, Haifeng Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating Vulnerability Assessment through PoC Migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) (CCS '21). Association for Computing Machinery, New York, NY, USA, 3300–3317. doi:10.1145/3460120.3484594
- [14] Brett Daniel, Danny Dig, Tihomir Gvero, Vilas Jagannath, Johnston Jia, Damion Mitchell, Jurand Nogiec, Shin Hwei Tan, and Darko Marinov. 2011. ReAssert: a tool for repairing broken unit tests. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) (ICSE '11). Association for Computing Machinery, New York, NY, USA, 1010–1012. doi:10.1145/1985793.1985978
- [15] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. 2009. ReAssert: Suggesting Repairs for Broken Unit Tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, 433–444. doi:10.1109/ASE.2009.17
- [16] National Vulnerability Database. [n. d.]. *CVE-2024-22257*. <https://nvd.nist.gov/vuln/detail/cve-2024-22257>
- [17] Peng Deng, Lei Zhang, Yuchuan Meng, Zhemin Yang, Yuan Zhang, and Min Yang. 2025. CHAINFUZZ: exploiting upstream vulnerabilities in open-source supply chains. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) (SEC '25). USENIX Association, USA, Article 319, 20 pages.
- [18] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the detection of inconsistencies in public security vulnerability reports. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) (SEC'19). USENIX Association, USA, 869–885.
- [19] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 204–215. doi:10.1145/3293882.3303057
- [20] Yi Gao, Xing Hu, Zirui Chen, and Xiaohu Yang. 2025. Vulnerability-Triggering Test Case Generation from Third-Party Libraries. arXiv:2409.16701 [cs.SE] <https://arxiv.org/abs/2409.16701>
- [21] Konstantin Grotov, Sergey Titov, Yaroslav Zharov, and Timofey Bryksin. 2024. Untangling Knots: Leveraging LLM for Error Resolution in Computational Notebooks. arXiv:2405.01559 [cs.SE] <https://arxiv.org/abs/2405.01559>
- [22] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) (ESEC/FSE 2021). Association for Computing Machinery, New York, NY, USA, 478–490. doi:10.1145/3468264.3468571
- [23] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. *IEEE Trans. Softw. Eng.* 49, 8 (Aug. 2023), 4004–4022. doi:10.1109/TSE.2023.3278129
- [24] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. 2009. Semantics-aware trace analysis. *SIGPLAN Not.* 44, 6 (June 2009), 453–464. doi:10.1145/1543135.1542527
- [25] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *2012 IEEE Symposium on Security and Privacy*, 48–62. doi:10.1109/SP.2012.13
- [26] Abbas Javan Jafari, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. 2023. Dependency Update Strategies and Package Characteristics. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 149 (Sept. 2023), 29 pages. doi:10.1145/3603110
- [27] JetBrains. [n. d.]. *Website of IDEA*. <https://www.jetbrains.com/idea/>
- [28] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaoming Tang, Manuel Egele, Chao Zhang, and Mathias Payer. 2022. Evocatio: Conjuring Bug Capabilities from a Single PoC. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) (CCS '22). Association for Computing Machinery, New York, NY, USA, 1599–1613. doi:10.1145/3548606.3560575
- [29] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. 2023. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2122–2137. doi:10.1109/SP46215.2023.1017928
- [30] Hyeonseong Jo, Jinwoo Kim, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. 2021. GapFinder: Finding Inconsistency of Security Information From Unstructured Text. *IEEE Transactions on Information Forensics and Security* 16 (2021), 86–99. doi:10.1109/TIFS.2020.3003570
- [31] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S. Păsăreanu, and David Lo. 2022. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) (ISSTA 2022). Association for Computing Machinery, New York, NY, USA, 276–288. doi:10.1145/3533767.3534398
- [32] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018), 384–417.
- [33] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018), 384–417.
- [34] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2024. Poison Attack and Poison Detection on Deep Source Code Processing Models. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 62 (March 2024), 31 pages. doi:10.1145/3630008
- [35] Siyuan Li, Yongpan Wang, Chaopeng Dong, Shouguo Yang, Hong Li, Hao Sun, Zhe Lang, Zuxin Chen, Weijie Wang, Hongsong Zhu, and Limin Sun. 2023. LibAM: An Area Matching Framework for Detecting Third-Party Libraries in Binaries. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 52 (Dec. 2023), 35 pages. doi:10.1145/3625294
- [36] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-Preserving Test Repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 217–227. doi:10.1109/ICST.2019.00030
- [37] Shuhan Liu, Jiayuan Zhou, Xing Hu, Filipe Roseiro Cogo, Xin Xia, and Xiaohu Yang. 2025. An Empirical Study on Vulnerability Disclosure Management of Open Source Software Systems. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 214 (Aug. 2025), 31 pages. doi:10.1145/3716822
- [38] Looly. [n. d.]. *Vulnerability Introduce Commit for CVE-2023-51080*. <https://github.com/chinabugotech/hutool/commit/c45b3f>
- [39] Mehdi Mirzaaghaei. 2011. Automatic test suite evolution. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) (ESEC/FSE '11). Association for Computing Machinery, New York, NY, USA, 396–399. doi:10.1145/2025113.2025172
- [40] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2014. Automatic test case evolution. *Softw. Test. Verif. Reliab.* 24, 5 (Aug. 2014), 386–411. doi:10.1002/stvr.1527
- [41] 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 *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) (SEC '18). USENIX Association, USA, 919–936.
- [42] Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shaping Li. 2023. Fine-grained Commit-level Vulnerability Type Prediction by CWE Tree Structure. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 957–969. doi:10.1109/ICSE48619.2023.00088
- [43] Shengyi Pan, Lingfeng Bao, Jiayuan Zhou, Xing Hu, Xin Xia, and Shaping Li. 2024. Towards More Practical Automation of Vulnerability Assessment. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 148, 13 pages. doi:10.1145/3597503.3639110

- [44] Shengyi Pan, You Wang, Zhongxin Liu, Xing Hu, Xin Xia, and Shaping Li. 2024. Automating Zero-Shot Patch Porting for Hard Forks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 363–375. doi:10.1145/3650212.3652134
- [45] Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Cogo, Xin Xia, Lingfeng Bao, Xing Hu, Shaping Li, and Ahmed E. Hassan. 2022. Automated unearthing of dangerous issue reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 834–846. doi:10.1145/3540250.3549156
- [46] Shanto Rahman, Sachit Kuhar, Berk Cirisci, Pranav Garg, Shiqi Wang, Xiaofei Ma, Anoop Deoras, and Baishakhi Ray. 2025. UTFix: Change Aware Unit Test Repairing using LLM. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 85 (April 2025), 26 pages. doi:10.1145/3720419
- [47] Shanto Rahman and August Shi. 2024. FlakeSync: Automatically Repairing Async Flaky Tests. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE ‘24). Association for Computing Machinery, New York, NY, USA, Article 136, 12 pages. doi:10.1145/3597503.3639115
- [48] Ahmadreza Saboor Yaragh, Darren Holden, Nafiseh Kahani, and Lionel Briand. 2025. Automated Test Case Repair Using Language Models. *IEEE Trans. Softw. Eng.* 51, 4 (Feb. 2025), 1104–1133. doi:10.1109/TSE.2025.3541166
- [49] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, and Min Yang. 2023. Precise (Un)Affected Version Analysis for Web Vulnerabilities. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) (ASE ‘22). Association for Computing Machinery, New York, NY, USA, Article 76, 13 pages. doi:10.1145/3551349.3556933
- [50] Joseph Spracklen, Raveen Wijewickrama, A H M Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jalliwala. 2025. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. arXiv:2406.10279 [cs.SE] <https://arxiv.org/abs/2406.10279>
- [51] Synopsys. [n. d.]. OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT 2023. <https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html>
- [52] Tekul. [n. d.]. Spring Security Commit. <https://github.com/spring-projects/spring-security/commit/76438b>
- [53] Trustin. [n. d.]. Vulnerability Introduce Commit for CVE-2021-43795. <https://github.com/line/armeria/commit/e2697a575e9df6692b423e02d731f293c1313284>
- [54] Gladys Tyen, Hassan Mansoor, Victor Cărbune, Peter Chen, and Tony Mak. 2024. LLMs cannot find reasoning errors, but can correct them given the error location. arXiv:2311.08516 [cs.AI] <https://arxiv.org/abs/2311.08516>
- [55] Vision-version. [n. d.]. Exploit for CVE-2020-5245. <https://github.com/vision-version/vision-version.github.io/blob/main/Vision/1/groundtruth/testcase-trigger/testcase-trigger/CVE-2020-5245/pom.xml>
- [56] Siqi Wang, Xing Hu, Bei Wang, Wenxin Yao, Xin Xia, and Xinyu Wang. 2025. Refactoring Deep Learning Code: A Study of Practices and Unsatisfied Tool Needs. In *International Conference on Software Maintenance and Evolution*.
- [57] Susheng Wu, Ruisi Wang, Kaifeng Huang, Yiheng Cao, Wenyang Song, Zhuotong Zhou, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Vision: Identifying Affected Library Versions for Open Source Software Vulnerabilities. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) (ASE ‘24). Association for Computing Machinery, New York, NY, USA, 1447–1459. doi:10.1145/3691620.3695516
- [58] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 335–346. doi:10.1109/ICPC.2019.00052
- [59] Yong Xu, Bo Huang, Guoqing Wu, and Mengting Yuan. 2014. Using Genetic Algorithms to Repair JUnit Test Cases. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 1. 287–294. doi:10.1109/APSEC.2014.51
- [60] Zhipeng Xue, Zhipeng Gao, Xing Hu, and Shaping Li. 2023. ACWRecommender: A Tool for Validating Actionable Warnings with Weak Supervision. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1876–1880.
- [61] Zhipeng Xue, Zhipeng Gao, Shaohua Wang, Xing Hu, Xin Xia, and Shaping Li. 2024. Selfpic: Self-guided partial code execution with llms. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1389–1401.
- [62] Yanming Yang, Xing Hu, Zhipeng Gao, Jinfu Chen, Chao Ni, Xin Xia, and David Lo. 2024. Federated Learning for Software Engineering: A Case Study of Code Clone Detection and Defect Prediction. *IEEE Transactions on Software Engineering* 50, 2 (2024), 296–321. doi:10.1109/TSE.2023.3347898
- [63] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE116 (June 2025), 23 pages. doi:10.1145/3729386
- [64] Qi Zhan, Xing Hu, Zhiyong Li, Xin Xia, David Lo, and Shaping Li. 2024. PS3: Precise Patch Presence Test based on Semantic Symbolic Signature. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE ‘24). Association for Computing Machinery, New York, NY, USA, Article 167, 12 pages. doi:10.1145/3597503.3639134
- [65] Qi Zhan, Xing Hu, Xin Xia, and Shaping Li. 2024. REACT: IR-Level Patch Presence Test for Binary. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) (ASE ‘24). Association for Computing Machinery, New York, NY, USA, 381–392. doi:10.1145/3691620.3695012
- [66] Zheng Zhang, Yu Hao, Weiteng Chen, Xiaochen Zou, Xingyu Li, Haonan Li, Yizhuo Zhai, Zhiyun Qian, and Billy Lau. 2024. SymBisect: accurate bisection for fuzzer-exposed vulnerabilities. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) (SEC ‘24). USENIX Association, USA, Article 140, 18 pages.
- [67] Hao Zhong and Na Meng. 2024. Compiler-directed Migrating API Callsite of Client Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE ‘24). Association for Computing Machinery, New York, NY, USA, Article 226, 12 pages. doi:10.1145/3597503.3639084
- [68] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E. Hassan. 2023. CoLeFunDa: Explainable Silent Vulnerability Fix Identification. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (ICSE ‘23). IEEE Press, 2565–2577. doi:10.1109/ICSE48619.2023.00024
- [69] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E. Hassan. 2022. Finding a needle in a haystack: automated mining of silent vulnerability fixes. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) (ASE ‘21). IEEE Press, 705–716. doi:10.1109/ASE51524.2021.9678720
- [70] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) (ASE ‘24). Association for Computing Machinery, New York, NY, USA, 1633–1644. doi:10.1145/3691620.3695531
- [71] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 995–1010.
- [72] Anni Zou, Wenhao Yu, Hongming Zhang, Kaixin Ma, Deng Cai, Zhuosheng Zhang, Hai Zhao, and Dong Yu. 2024. DOCBENCH: A Benchmark for Evaluating LLM-based Document Reading Systems. arXiv:2407.10701 [cs.CL] <https://arxiv.org/abs/2407.10701>
- [73] Xiaochen Zou, Yu Hao, Zheng Zhang, Juefei Pu, Weiteng Chen, and Zhiyun Qian. 2024. SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem. *NDSS* (2024). doi:10.14722/ndss.2024.24926