Accurate Identification of the Vulnerability-Introducing Commit based on Differential Analysis of Patching Patterns

Qixuan Guo
School of Cyberspace Security
Beijing Jiaotong University, China
gqxuan@bjtu.edu.cn

Yongzhong He*
School of Cyberspace Security
Beijing Jiaotong University, China
yzhhe@bjtu.edu.cn

***Abstract*—When a vulnerability is detected in a specific software version, it is critical to trace the commit history to accurately identify the first commit where the vulnerability was introduced, known as Vulnerability-Introducing Commit(VIC). This article proposes a method to accurately identify the VIC based on differential analysis of vulnerability patching patterns. Firstly, we compare the two files, before and after patching a vulnerability, to classify vulnerability-related statements in the patch into different patching patterns, such as coding errors, improper data flow, misplaced statements, and missing critical checks. Then, based on the patching patterns, we extract a vulnerability-critical statement sequence from the vulnerable file and match it with the earlier commits to determine the introducing commit. To evaluate the effectiveness of this method, we collected a dataset comprising 6,920 CVEs and 5,859,238 commits from open-source software, including the Linux kernel, MySQL, and OpenSSL, etc. The experimental results demonstrate that the proposed method achieves a detection accuracy of 94.94% and a recall rate of 86.92%, significantly outperforming existing approaches.**

I. Introduction

To mitigate the risks posed by vulnerabilities in source code files, various approaches are employed, including code reviews, automated vulnerability scans, penetration tests, and timely software patches. However, a study by Blackduck [1] revealed that, in larger companies, an average of 41% of vulnerabilities identified within 12 months remain unpatched and unresolved, leaving them susceptible to exploitation by adversaries.

The failure to patch identified vulnerabilities can be attributed to several factors, including the lack of available patches or inadequate technical support for the applications. One of the challenges faced by maintenance teams is determining whether a specific version is unaffected by the vulnerability or if the patch is unsuitable for that version, particularly when patching attempts fail. Currently, public vulnerability databases, such as the **N**ational **V**ulnerability **D**atabase (NVD) [2], the **O**pen **S**ource **V**ulnerability **D**atabase (OSVDB) [3], and Bugtraq [4], record the affected versions of vulnerabilities. For example, the National Vulnerability Database (NVD) provides detailed data on **C**ommon **V**ulnerabilities and **E**xposures (CVEs), including their descriptions and affected versions of software. At the same time, commercial security services (such as Hakiri [5], Snyk [6], and SourceClear [7]) and open source security tools (such as BundlerAudit [8], OWASP OSSIndex [9], and Dependency-Check [10]) rely on vulnerability information from the NVD to operate effectively.

However, the information in the database is not reliable. It is shown that approximately 25% of the versions were mistakenly marked as affected versions [11]. The error rate is significant and cannot be overlooked. Dong et al. [12] developed a system to analyze 20 years of data, revealing that inconsistencies in vulnerable software versions are widespread, with only 59.82% of vulnerability reports fully matching the NVD. Other works [13], [14] have also identified some data quality problems with the NVD.

To decide whether a software version is affected by a specific vulnerability, it is essential to trace the earlier commits to identify the first commit, namely **V**ulnerability-**I**ntroducing **C**ommit(**VIC**), where the vulnerability was introduced. There are several studies for the identification of VIC. For example, B-SZZ [15] and V-SZZ [16] are based on the deleted statements in patches to trace the VIC, but other statements that are critical to the vulnerability are ignored. Manually identifying dangerous functions and building the complete paths that trigger vulnerabilities can precisely detect vulnerable files [17]. However, this approach lacks scalability and requires a significant amount of expertise. This article proposes an efficient and accurate method to identify the VIC based on a refined patch analysis.

Challenges. A Security patch contains important information about a vulnerability, so the patch is used to decide whether a CVE exists or not by patch presence detection. There are three challenges in identifying the VIC based on patches. Firstly, patches are often specifically developed for the software version in which a vulnerability is discovered. The vulnerability may have been introduced into the software project in an earlier version than the one in which it was discovered, meaning that earlier versions may be vulnerable. During software development, each software version underwent numerous updates and changes, including feature

* Corresponding Author

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230140
www.ndss-symposium.org

additions, performance improvements, and code refactoring. The patch probably does not match the earlier version of the vulnerable files exactly. Therefore, it is unsafe to determine the VIC solely based on an exact comparison with the patch. Secondly, not all statements in the patch are related to a vulnerability. For example, some statements are used as positioning or context for the changed statements, while others are used for styling and formatting adjustments. If we include these statements in identifying the VIC, the result is error-prone because a mismatch of these statements does not deny the existence of the vulnerability. Lastly, not all vulnerability-critical statements are included in the patch. For example, to fix a buffer overflow, a condition check is added to verify the boundary. However, the statement accessing the buffer, which is essential to trigger the vulnerability, is not contained in the patch. If these important statements are not used in the VIC identification method, their changes are not be taken into consideration, which degrades the precision of the method. Therefore, it is challenging to automatically extract the statements most closely related to the vulnerability based on patch analysis.

Approach. Our key insight is that the insertion and deletion statements contained in a patch are not independent, and their relationship is crucial in revealing the logic of vulnerability and identifying statements critical to a vulnerability. For example, some insertion statements are used as replacements for certain deletion statements or move statements to other positions. We propose a differential analysis of vulnerability patches, files, and their commits to capture the relationship between statements and accurately identify the VIC. First, we filter and remove the noise statements that are obviously not relevant to the vulnerability. Based on our observations of the patches, we select vulnerability noise statement types that are apparent and frequently exist in the patches, and perform filtering and removal of these statements based on type features. Second, we extract the patching patterns. We compare the control flows of two files, immediately before and after patching a vulnerability, and classify vulnerability-related statements in the patch based on their differences in statement location, operator, and parameters into different patching patterns, such as coding errors, improper data flow, misplaced statements, and missing critical checks. Then, based on the patching patterns, we extract a vulnerability-critical statement sequence from the vulnerable file. Lastly, we match the vulnerability-critical statement sequence with the earlier commits to determine the introducing commit.

Contributions. Our contributions are listed below.

(1) We propose an approach to extract the vulnerability-critical statement sequence by differential analysis of patching patterns. By focusing on the differences in statements location, operator, and parameters in the control flows of vulnerable and patched files, we can capture the nuances of patching semantics and accurately generate the vulnerability-critical statement sequence.

(2) We implemented a tool, VicDiff, to identify the VIC. Our tool relies on lightweight static analysis of patches and commits, allowing it to be automated and scalable. VicDiff effectively and accurately identifies the VICs on a large scale.

(3) We collect a data set comprising 6,920 CVEs and 5,859,238 commits from open-source software such as the Linux kernel, MySQL, and OpenSSL. The experimental results demonstrate that the proposed method achieves a detection accuracy of 94.94% and a recall rate of 86.92%, significantly outperforming existing approaches.

II. Background

In this section, we provide the background for an easy understanding of our patch-based VIC detection.

Commit and Security patch. A commit updates a set of files in a software project and produces a set of new files at a specific point in time within a version control system for software development. A security patch is a special commit to files that fixes a security vulnerability. Each known vulnerability has a unique CVE ID [18], which is assigned by the MITRE Corporation [19].

Each commit comprises modifications to multiple files, where the changes in each file consist of multiple diff blocks. The diff block [20] is structured into sections, in which the diff section contains: lines prefixed with "+" (which represent statements inserted into the files, called $'Insertion\ Statements'$), lines prefixed with "-" (which indicates statements deleted from the file, called $'Deletion\ Statements'$), and lines without any prefix (which represent the unchanged statements surrounding the inserted or deleted statements).

Security Patch type. Security patches can be classified into three types. Insertion-Only Patch($InsPatch$): A patch of this type only includes $Insertion\ Statements$ without $Deletion\ Statement$. For example, to fix a buffer overflow vulnerability, a check statement may be inserted into the boundary only before the memory access operation without deleting any statements from vulnerable files. Deletion-Only Patch($DelPatch$): A patch of this type only includes $Deletion\ Statements$ and without any $Insertion\ Statement$. For example, to fix a use-after-free vulnerability, one can simply remove the statement that refers to a data structure that is released by an old instance. Insertion and Deletion Patch($InsDelPatch$): A patch of this type includes both $Insertion\ Statements$ and $Deletion\ Statements$, typically used to replace existing flawed statements with new statements. For example, to fix an integer overflow vulnerability, it may modify the flawed statement by adding a forced type conversion. To accomplish this modification, the flawed statement is deleted and the corrected statement is inserted in the patch.

CVE Description and Gitlog/Changelog. CVE descriptions typically include the vulnerability type (e.g., buffer overflow, use-after-free), the location of the affected code (such as file paths and function modules), and an analysis of the trigger mechanism. Additionally, git log and changelog provide summaries of key code changes, including the root cause and the fix for the vulnerability. Several Git repository

websites, such as the Linux kernel, also indicate the commit where the vulnerability was introduced. This information helps researchers better understand the origin and remediation of vulnerabilities.

III. Definitions

A. Problem Definition

When a vulnerability is detected in a specific software version, determining which versions are affected is a crucial issue for security maintenance. To resolve this issue, it is essential to trace the history of commits to identify the VIC in which the vulnerability was first introduced into the software project. A VIC is the turning point of the vulnerability's existence, before which the vulnerability does not exist, and after which the vulnerability is exploitable. We organize the commits related to a specific source code file in reverse chronological order, notated as $Com_1, Com_2, ..., Com_i, Com_j, ..., Com_n$, where Com_1 represents the commit where the vulnerability was detected and fixed, Com_i denotes the commit where the vulnerability was initially introduced. All versions of the specific source code file are noted, respectively, as $File_1, File_2, ..., File_i, File_j, ..., File_n$, such that $File_i$ is produced by commit Com_i to $File_{i+1}$. Com_1 is the security patch for a specific CVE, and $File_1$ is patched for the CVE vulnerability. $File_1$ is also denoted F_p, which means this file is patched, and $File_2$ is F_v, which means this file is vulnerable. The relationship between commits and files is illustrated in Figure 1.

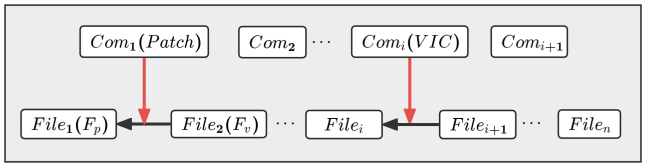

Fig. 1. The relationship between commits and files.

B. Definitions of Terminology

Noise Statements. Patches often contain statements apparently unrelated to the vulnerability logic, which we denote as Noise Statements. Formally, a Noise Statement is a statement (including *Insertion Statements* and *Deletion Statements*) in the patch Com_1 if the statement belongs to a non-semantic modification type, such as function renaming, comment lines, formatting changes, etc. Noise statements are commonly used for code refactoring, where developers optimize the code structure, enhance readability, and improve maintainability by simplifying complex logic, reorganizing functions or classes, and optimizing algorithms. Although such statements are apparently not related to a vulnerability, they will adversely affect the performance of the patch-based methods if they are not excluded.

Vulnerability-related Statements. Formally, vulnerability-related statements,$VRStmt$ for short, are a set of statements,

```
1 @@ -6520,6 +6520,12 @@ static int nft_setelem_deactivate(const struct net *net,
2 +static void nft_setelem_catchall_destroy(struct nft_set_elem_catchall *catchall)
3 +{
4 +    list_del_rcu(&catchall ->list);
5 +    kfree_rcu(catchall, rcu);
6 +}
7 +
8 @@ -6528,8 +6534,7 @@ static void nft_setelem_catchall_remove (const struct net *net,
9     if (catchall->elem == elem_priv) {
10 -        list_del_rcu(&catchall->list);
11 -        kfree_rcu(catchall, rcu);
12 +        nft_setelem_catchall_destroy(catchall);
13          break;
14 @@ -9678,11 +9683,12 @@ static struct nft_trans_gc *nft_trans_gc_catchall(struct nft_trans_gc *gc,
15 - struct nft_set_elem_catchall *catchall;
16 + struct nft_set_elem_catchall *catchall, *next;
17     const struct nft_set *set = gc->set;
18 + struct nft_elem_priv *elem_priv;
19     struct nft_set_ext *ext;
20
21 - list_for_each_entry_rcu(catchall, &set->catchall_list, list) {
22 + list_for_each_entry_safe(catchall, next, &set->catchall_list, list) {
23          ext = nft_set_elem_ext(set, catchall->elem);
24 @@ -9700,7 +9706,13 @@ dead_elem:
25     if (sync)
26         gc = nft_trans_gc_queue_sync(gc, GFP_ATOMIC);
27     else
28         gc = nft_trans_gc_queue_async(gc, gc_seq, GFP_ATOMIC);
29     if (!gc)
30         return NULL;
31 -   nft_trans_gc_elem_add(gc, catchall->elem);
32 +   elem_priv = catchall->elem;
33 +   if (sync) {
34 +       nft_setelem_data_deactivate(gc->net, gc->set, elem_priv);
35 +       nft_setelem_catchall_destroy(catchall);
36 +   }
37 +   nft_trans_gc_elem_add(gc, elem_priv);
38 }
39 return gc;
```

Fig. 2. The patch of the running example CVE-2023-6111.

including all *Insertion Statements* and *Deletion Statements* in a patch Com_1 with subtraction of the set of noise statements. There is a possibility that some statements in $VRStmt$ are not used to fix a vulnerability, but rather to add a new feature. However, based on our observations, this is very rare in our datasets.

Differential Analysis & Patching Patterns. Differential analysis is the key method to classify the statements in $VRStmt$ into different categories (Patching Patterns) based on the characteristics of this statement in the **C**ontrol **F**low **G**raph(CFG). The method is based on an analysis of the differences in location, operator, and parameters between the two CFGs of the vulnerable and patched files, which flow across each statement in $VRStmt$. After this analysis, every statement in vulnerability-related statements $VRStmt$ is assigned

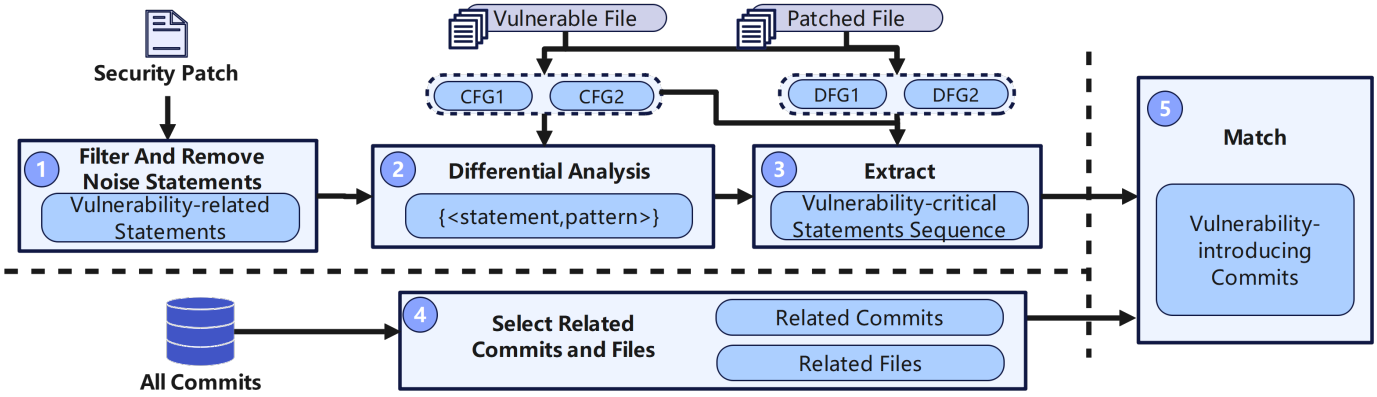

Fig. 3. The workflow of our approach.

a pattern label, denoted as a pair: $< statement, pattern >$.

Vulnerability-critical Statement Sequence. Vulnerability-critical Statement Sequence(denoted as $VCSeq$) is defined as a set of statements from the vulnerable source file, and arranged in sequence, based on statement patterns after the analysis of $VRStmt$, the CFGs, and the **D**ata **F**low **G**raph(DFG) of the vulnerable file F_v. We note that 1) the statements in $VCSeq$ are from the vulnerable source file F_v, but may not be from the patches, and 2) the order of the statements in the sequence should be the same as that in F_v.

C. A Running Example

We present a running example of CVE-2023-6111 [21] and its patch to illustrate our motivation and approach. The snippet of statements is shown in Figure 2 and clearly shows the challenges in existing studies. This vulnerability is a typical use-after-free (UAF) vulnerability. Specifically, when the function *nft_trans_gc_catchall* is invoked with the parameter sync set to true, it fails to remove the catchall set element from the *catchall_list*. This oversight allows the same element to be released multiple times, leading to a use-after-free condition. This vulnerability illustrates the following challenges: (1) Not all statements in a patch are directly related to fixing the vulnerability. Lines 2–7 restructure the logic from lines 10–11 to reduce duplication: These are structural optimizations, not part of the actual fix. The variable declarations in lines 15, 16, and 18 are added or removed to support later patch logic but do not impact the vulnerability itself. Including such statements can interfere with accurately identifying the VIC. (2) Not all statements related to the vulnerability are included in the patch. In this case, the vulnerability is triggered when the parameter sync is true and the element of the catchall set fails to be removed from *catchall_list* in a timely manner, leading to its repeated release. The patch fixes this vulnerability by calling the functions *nft_setelem_data_deactivate()* and *nft_setelem_catchall_destroy()* to remove the element. Therefore, in the absence of these two function calls (on lines 34 and 35 of the patch), the vulnerability will still occur as long as lines 25 and 30 exist. As a result, although line 26 was not modified in this patch, it serves as a critical trigger for the vulnerability.

IV. Our Approach

We propose an approach to identify the VIC on a large scale automatically. The workflow of our approach is shown in Figure 3.

A. Workflow of our approach

- Filtering and Removal of Noise Statements. Takes the CVE patch as input and removes non-functional code (noise statements), outputting a refined set of vulnerability-related statements, denoted as $VRStmt$.
- Differential Analysis. Uses $VRStmt$, along with the vulnerable and patched versions (F_v and F_p), to classify statements based on control flow analysis. The output is a set of pairs: $\{< statement, pattern >\}$.
- Extraction. Combines control flow, data flow, and patching patterns from the previous output and F_v and F_p to derive $VCSeq$.
- Selection of Related Commits and Files. Analyzes the full commit history of the vulnerable file to identify and chronologically store commits and files related to the vulnerable function.
- Matching. Matches $VCSeq$ against the filtered files. The first commit where the match fails is identified as the VIC.

B. Filtering and Removal of Noise Statements

We filter the five most common types of noise statements. They are non-semantic modifications, defined as follows:

Function renaming: for readability or developer preference, with no impact on program behavior and vulnerability. However, function renaming can cause inconsistency in commit-based trace-back processes. To maintain accurate function tracking, it is necessary to record the mapping between the previous and the post name of the same function during analysis.

Extract method: for reusability of statements by extracting repetitive logic into separate functions, which complicates vulnerability analysis.

Variable declarations: not critical because the statements accessing these variables are critical and not filtered out.

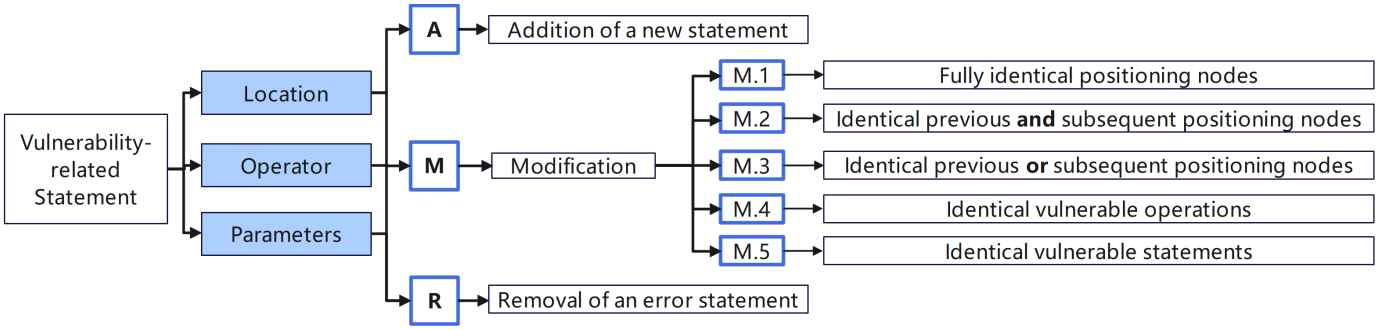

Fig. 4. Patching patterns of vulnerability-related statements

Comment lines: for human understanding only; irrelevant to execution.

Empty lines: for formatting purposes, but introduce noise in version comparisons.

The five types of noise statements in a patch are identified and removed by syntax analysis and regular expression matching. We do not filter all noise statements as it is challenging to precisely identify some of them, such as complicated refactoring statements. We selected five types of noise statements because they are very clearly defined, easy to detect, and frequently used. Meanwhile, according to our study, other types, such as complicated refactoring or adding new features in a security patch, are rare. We illustrate this step with the running example. The noise statements in CVE-2023-6111 patch are filtered out: the variable declarations in lines 15, 16, and 17 are filtered and removed. The statements related to the extract method in lines 2-6, as well as in lines 10, 11, and 12, are filtered out. The blank line, e.g. line 7 will be filtered out. $VRStmt$ consists of the remaining $InsertionStatements$ and $DeletionStatements$. Filtering out noise statements ensures that the analysis focuses on meaningful code changes, thereby improving the accuracy of our approach.

C. Differential Analysis

The purpose of this step is to analyze the relationship between insertion and deletion statements to classify vulnerability-related statements into patching patterns. The idea of patching patterns was motivated by our observation of different patching behaviors in patches and their various consequences to vulnerability critical logic. Patching a file generally includes the following three behaviors: addition, removal, or modification of a statement.

So we first define three patterns: Pattern A, Pattern R, and Pattern M, as shown in Figure 4.

Removal of an error statement (**Pattern R**) Pattern R involves removing unnecessary or erroneous statements to eliminate potential vulnerabilities or errors. This typically addresses logical errors or redundant operations in the original statements, where simply deleting the faulty statement resolves the issue.

Addition of a new statement (**Pattern A**) Pattern A statements are often used to limit access or add other security checks. If the program lacks specific logic or checks, the patch supplements these omissions by inserting a boundary check statement.

Modification (**Pattern M**) Pattern M involves more complex modifications, typically requiring the deletion of erroneous statements followed by the insertion of correct statements. The application of Pattern M usually occurs when there are fundamental flaws in the original statements, and the functionality is repaired or improved by replacing erroneous logic.

It is apparent that all statements in $InsPatch$ are Pattern A, and all statements in $DelPatch$ are Pattern R, because for Pattern M, there are insertion statements to replace deletion statements in the same patch. On the contrary, it is more difficult to classify a statement in a $InsDelPatch$ into one of the above three patterns. Furthermore, the statements of Pattern M exhibit different patching behaviors and consequences for vulnerability, necessitating further refinement of their classification. The modification statement includes modifying the content of a statement without changing its location, modifying its location without changing its content, etc. The challenge in classifying such a statement is that there is no information describing the type of the statement or the relationship between the statements in a patch. To overcome this challenge, we propose classifying patching behaviors into patterns based on whether the location and content (operator or parameters) are wholly or partially modified. We emulate all of them, then combine some into a single pattern based on the semantic analysis of patching behaviors across a large scale of patches. We also remove types without practical meanings (for example, those where both content and position remain unchanged, as they are meaningless for patching). Consequently, we defined seven patching patterns for statements in a $InsDelPatch$ so that every pattern corresponds to a set of real-world patching instances. Each statement in a patch can be assigned to one of the patching patterns based on the information provided within the patch and its related source code files.

To define the refined patterns formally, we first introduce some notation. The set of $Insertion\ Statements$ is denoted as I, and the set of $Deletion\ Statements$ is denoted as D. Each statement in D is represented as D_i, where $D_i \in D$. Similarly, each statement in I is represented as I_j, where $I_j \in I$. Here, i and j represent the respective positions of

the statement patterns in the sets F_p and F_v.

By generating the CFGs of the functions in F_p and F_v using the aforementioned method, we can obtain the node information for D_i and I_j, along with their complete control flow and data flow. Since there may be more than one flow through D_i and I_j in F_p and F_v, we preserve all control flow information passing through F_p and F_v.

Taking a code snippet line 25-31, 39 from the vulnerability function of CVE-2023-6111 in Figure 2 as an example, its CFG is shown in Figure 5. Line 31 (*nft_trans_gc_elem_add*) is a vulnerability-related statement. One control flow path is: line 25 (if (sync)), line 26, 29, 31, 39. Another control flow path is: line 25 (if (sync)), line 28, 29, 31, 39. In both paths, except for the node itself (line 31), the remaining five statements are considered **positioning nodes**.

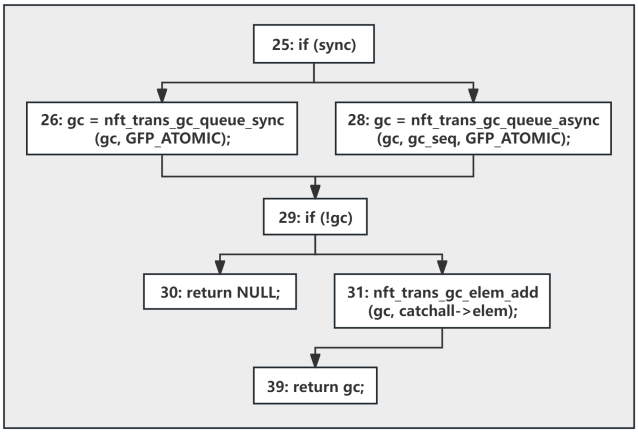

Fig. 5. The partial CFG of CVE-2023-6111.

For each pair of vulnerability-related statements D_i and I_j in the patch, we compare their locations, operators, and parameters. Through an iterative comparison, vulnerability-related statements are classified. Based on the results of the comparison, these vulnerability-related statements are categorized into patching patterns as follows. First, the locations of D_i and I_j are compared, specifically examining whether there are identical or partially identical positioning nodes between D_i and I_j.

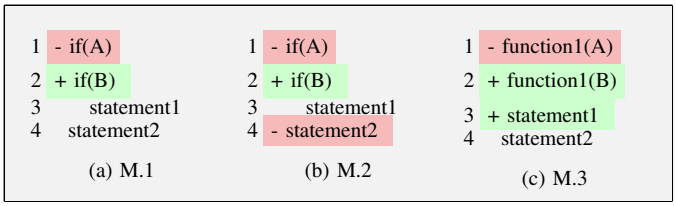

Fig. 6. Schematic diagram of Pattern M.1,M.2,M.3.

Fully identical positioning nodes(Pattern M.1) For D_i and I_j, if there exist control flows such that all positioning nodes between the statements are identical, except D_i is different from I_j, we define statements D_i and I_j as Pattern M.1. This pattern typically indicates a direct correction of one statement, while the overall logic remains unchanged. As shown in the code snippet in Figure 6.a, the modifications in lines 1 and 2 exhibit the characteristics of Pattern M.1.

Identical previous and subsequent positioning nodes (Pattern M.2) If there exist two control flows such that adjacent positioning nodes of D_i and I_j are identical (for example, the pair of direct previous nodes and the pair of direct subsequent nodes are identical), then the statements are classified as Pattern M.2. This pattern indicates that the previous and subsequent positioning nodes remain unchanged, with only the content of the vulnerability-related statements modified. As shown in the code snippet in Figure 6.b, the modifications in lines 1 and 2 satisfy the characteristics of Pattern M.2.

Identical previous or subsequent positioning nodes (Pattern M.3) If there exist two control flows such that the previous or subsequent positioning nodes of D_i or I_j are identical(but not both), and D_i and I_j have the same control statement keyword, the statements are classified as Pattern M.3. D_i and I_j have only partial similarity in their positioning nodes, but perform the same operator. This pattern suggests that while the new logic differs in implementation, the functional intent remains consistent. As shown in the code snippet in Figure 6.c, the modifications in lines 1 and 2 satisfy the characteristics of Pattern M.3.

Suppose I_j and D_i do not meet any of the above criteria. In that case, a further analysis of their operator and parameters will be performed. We analyze the I_j and D_i by comparing the operator and parameters. Based on the comparison results, they are classified as follows:

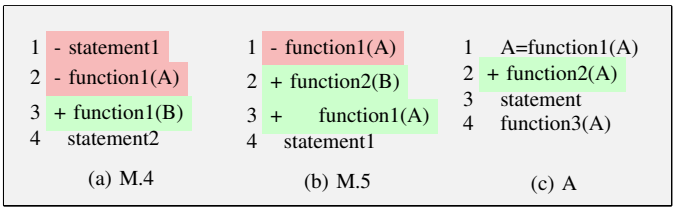

Fig. 7. Schematic diagram of Pattern M.4, M.5,A.

Identical vulnerable operators (Pattern M.4) Suppose there exist two control flows such that D_i and I_j have the same operators but different parameters. In that case, the statements are classified as Pattern M.4. Such changes usually indicate that developers aim to optimize or adjust specific input conditions while maintaining the original operator call, thus improving overall program execution. As shown in the code snippet in Figure 7.a, the modifications in lines 2 and 3 satisfy the characteristics of Pattern M.4.

Identical vulnerable statements (Pattern M.5) If there exist two control flows such that D_i and I_j have identical vulnerable statements, the statements are classified as Pattern M.5. This pattern may indicate that the developer needs to apply the same logic at different positioning nodes, which is common in handling conditional branches or loops. As shown in the code snippet in Figure 7.b, the modifications in lines 1 and 3 satisfy the characteristics of Pattern M.5.

If D_i and I_j do not meet any of the above conditions, we classify D_i as Pattern R and I_j as Pattern A.

The algorithm for classifying vulnerability-related statements is described as follows, based on the definition of patching patterns. Since $InsPatch$ contains only insertion statements, which match the characteristics of Pattern A, we can directly classify the insertion statements in the categorized $InsPatch$ as Pattern A. Similarly, $DelPatch$ contains only deletion statements, which match the characteristics of Pattern R, and thus the deletion statements in the categorized $DelPatch$ can be directly classified as Pattern R. This classification approach allows us to reduce unnecessary analysis work for $InsPatch$ and $DelPatch$, thereby improving the efficiency of analysis. Because the insertion statements and the deletion statements may have complicated relationships, we use differential analysis to classify the statements in $InsDelPatch$. The algorithm 1 illustrates how to perform the classification of the deletion statements pattern. The input is a deletion statement from the patch, and the output is the deletion statement with a pattern label.

Algorithm 1 Differential Analysis.

Input: D_i and I
Output: $< D_i, pattern >$

```
set pa to NULL
for I_j ∈ I do
    if is_identical(pos(D_i),pos(I_j))
        pa = pa ∪ M.1
    elif is_identical(pre_pos(D_i),pre_pos(I_j))
        and is_identical(sub_pos(D_i),sub_pos(I_j))
            pa = pa ∪ M.2
    elif is_identical(sub_pos(D_i),sub_pos(I_j))
        and is_identical(operator(D_i),operator(I_j))
            pa = pa ∪ M.3
    elif is_identical(pre_pos(D_i),pre_pos(I_j))
        and is_identical(operator(D_i),operator(I_j))
            pa = pa ∪ M.3
    elif is_identical(operator(D_i),operator(I_j))
        and is_identical(parameters(D_i),parameters(I_j))
            pa = pa ∪ M.5
    elif is_identical(operator(D_i),operator(I_j))
        pa = pa ∪ M.4
    if (pa is NULL)
            pa = R
end for
pattern = select_first(pa)
return <D_i, pattern>
```

The algorithm iterates through the insertion statements within the same patch to determine whether the current deletion statement has a semantic association with any of them. Specifically, it examines the relationship between the deletion and insertion statements in terms of location, operator, and parameters. The function called *is_identical* is to evaluate the structural and semantic similarity between the two statements. If a match is found, the algorithm assigns the corresponding pattern label to the deletion statement. If none of the insertion statements satisfies the *is_identical* condition, the deletion statement is assigned Pattern R. After all insertion statements are iterated, the final pattern of the deletion statement is the one selected by the function *select_first*. This function selects the first pattern from a set of patterns ordered as M.1,2,3,5,4 and R. This ordering is based on the conditions of the patterns, the most stringent the first. The insertion statement *pairing(is_identical)* with the deletion statement is assigned the same pattern. We can use a similar algorithm to assign patterns to the remaining insertion statements.

As an example, the statements in $VRStmt$ of CVE-2023-6111 are categorized as follows: For $Deletion\ Statements$, we need to analyze line 21 and line 31. Because line 21 and line 22 share identical previous and subsequent positioning nodes, we categorize it as the Pattern M.2. The positioning nodes of line 31 have changed, but it calls the same function as line 37. The only difference lies in the data that flows through it, so we categorize it as Pattern M.4. For the $Insertion\ Statements$ of lines 32 to 36, because they are used to introduce new logic to conditionally deactivate and destroy resources based on the synchronization state, which aligns with the Pattern A type category, we classify them as Pattern A.

D. Extraction of the Vulnerability-critical Statement Sequence

We utilize the patching patterns of vulnerability-related statements to extract the vulnerability-critical statement sequence from the CFG and the DFG. Since the insertion statements do not exist in earlier commits, they are not included in $VCSeq$. However, these insertion statements play a crucial role in our analysis, as they help identify key statements both within and outside the patch.

Pattern A statements insert new statements I_j into the source code. This pattern often indicates missing logic in the program, which leads to potential vulnerabilities. I_j is used to fix the missing logic. We analyze the data flow through I_j, and search upstream for statements that modify the variable found in I_j, and downstream for statements that use the variable found in I_j, then add the statements that are nearest to I_j in the DFG into $VCSeq$. As shown in the code snippet in Figure 7.c, the list contains the data flow that passes through the third-line statement. Given that line 2 is of Pattern A, we need to find the upstream statement that modifies parameter A and is closest to line 2, which turns out to be the first line. Next, we need to find the downstream statement that uses parameter A and is closest to line 2, which turns out to be line 4. Finally, we add the statements of lines 1 and 4 to $VCSeq$.

Pattern R contains errors or defects; therefore, we include Pattern R statement D_i in $VCSeq$.

For Pattern M.1 statement D_i, control flow remains the same except for this specific node, whose content is modified. This indicates a coding error, making it a critical statement. Therefore, it is added to $VCSeq$. For Pattern M.2 and M.3

statements, the vulnerability function has multiple modifications. The M.2 and M.3 statements are similar to M.1 at the level of local code blocks. Thus, the M.2 and M.3 statements themselves are critical and are also added to $VCSeq$.

The pairing Pattern M.4 statements D_i and I_j invoke the same operator but with different parameters, suggesting that the functionality remains unchanged; however, the data flowing through may be inappropriate. We include D_i in $VCSeq$. At the same time, we apply the same processing as in Pattern A. We analyze the data flow through I_j and search upstream for statements modifying the variable found in I_j, and downstream for statements using this variable. We then add the statements nearest to I_j in the DFG to $VCSeq$.

Pattern M.5 statement D_i only changes in its location, which indicates that D_i itself is not defective, but its location may be inappropriate or the logical control dependencies are flawed. We analyze the control flow that goes through D_i, and add the adjacent statements (both preceding and subsequent) to D_i in the CFG to the vulnerability-critical statement sequence. As shown in the code snippet in Figure 7.b, line 3 is of type M.5. We need to add its adjacent control flows, which are lines 2 and 4, to $VCSeq$.

We remark that the $InsertionStatements$ in Pattern M.1-M.3 and M.5 are not used in the above process because their paired $DeletionStatements$, which brought the vulnerabilities during differential analysis, have been taken into consideration.

Based on the above steps, we effectively extract the vulnerability-critical statements.

For example, the $VCSeq$ of CVE-2023-6111 is extracted as follows: For Pattern A type statements, their associated data dependencies are added to $VCSeq$. In this case, that includes lines 26. For Pattern M.2 type statements, the statement itself is directly added to $VCSeq$. In this case, this includes line 21. For Pattern M.4 type statements, the statement itself is directly included, along with its preceding and subsequent data dependencies. In this case, that includes lines 31 and 37. However, since insertion statements do not appear in the vulnerable file, only line 31 is included in the $VCSeq$. Ultimately, the extracted $VCSeq$ for this vulnerability includes lines 21, 26, and 31.

E. Selection of Related Commits and Files

The input for the final step also requires filtering out vulnerability-related commit files. Not every commit has an impact on vulnerability. Therefore, it is essential to select vulnerability-related commits and files.

First, the function modified by each commit can be identified through the function declaration in the header of the hunk, and then we select those commits that involve vulnerable functions. Next, we download the source files that are generated by these commits.

A key detail to note in this step is that, in large systems, function declarations typically change over time, especially in cases of frequent updates and iterations, which increases the likelihood of name changes. To address this challenge, we designed a function declaration stack that can automatically record both the old and new names of the functions, ensuring that regardless of how the function name evolves, the original name can still be successfully matched, thus preventing potentially vulnerable commits from being overlooked.

F. Matching Files for Vulnerability

In this step, $VCSeq$ is compared with the related file versions to identify the commit that introduced the vulnerability. Note that the commits and the related source code file versions are arranged in reverse chronological order. As shown in the algorithm 2, the matching procedure searches for the first statement in the sequence $VCSeq$ within vulnerable function in the first target file (the latest source file version), starting from the beginning of the file; if found, then searches following statement in $VCSeq$ within the target file starting from the following statement after the previous matched statements in the target file, and so on. If all statements in $VCSeq$ are matched, the target file and its commit are labeled as vulnerable. Then repeats the above procedure with the second target file until it encounters a mismatch. The last vulnerable commit is the VIC. In this procedure, we adopt an exact matching approach in the search for critical statements because even minor changes to critical statements may change their semantics. The performance of similarity-based matching is sensitive to the similarity threshold, and tends to vary with different datasets.

Algorithm 2 Matching.

Input: $VCSeq$, $related\_commits$ and $related\_files$
Output: $Vuln\_files$

```
1:  for file ∈ related_files do
2:      function_codes=extract(file, function_name)
    //The current line index of the sequence to be matched.
3:      seq_idx = 0;
4:      for line ∈ function_codes do
5:          if compare(line, VCSeq[seq_idx])
6:              seq_idx += 1
    //Proceed to the next line of the sequence.
            //All statements have been successfully matched.
7:              if seq_idx == len(VCSeq)
8:                  Vuln_file = Vuln_file + file
9:      end for
10:     if seq_idx < len(VCSeq)
11:         return
12: end for
```

V. Implementation

We implemented our system using Python 3.10, taking advantage of its extensive library ecosystem for source code analysis. We implement our system with three main components: (1) Data Crawling, for efficient and automated acquisition of data; (2) Static Pattern Analysis, for filtering and removing noise statements, as well as performing differential analysis; (3) Identification, for extracting vulnerability-critical statement

sequences, selecting relevant commits and files, and matching vulnerability-related files.

A. Data Crawling Module

This module crawls CVE metadata, CVE-ids, commit-ids (for patches) from nvd.nist.gov, and patches, source code files, and history commit-ids from git.kernel.org for the Linux kernel. Our crawler utilizes the $lxml/etree$ library for extracting vulnerability-related content through efficient HTML/XML parsing and web scraping. To simplify the crawling process, we extract the patch commit-id from the URL with a different domain name in NVD (such as github.com/torvalds or git.kernel.org) and crawl all patches from git.kernel.org/pub/scm. For other projects with small CVE datasets, such as MySQL and OpenSSL, we use the NVD crawler to get CVE IDs, and then manually download other relevant data.

B. Static Pattern Analysis Module

This module classifies statements into corresponding patterns through filtering, removing noise statements, and performing differential analysis. This module utilizes the Python library re to define filtering rules and match each type of noise statement using regular expressions. Specifically, (1) Function renaming: identified by searching a neighboring delete-insert pair in the patch, where both lines have the format of function definition syntax. (2) Extract method: the block of insertion statements together are recognized as a new function by syntax matching. (3) Variable declarations: matched via variable declaration syntax analysis as (data type, variable name, semicolon). (4) Comments and empty lines: identified by comment symbols (e.g., //, /* */, #) or null content. After applying the filtering rules, the statements are stored in a list $VRStmt$. During the analysis of vulnerability files and patched files, we use the $pycparser/c\_parser$ and $pycparser/c\_ast$ libraries to parse the source code of the vulnerable function into an Abstract Syntax Tree (AST). The AST is converted into CFG using Python libraries $networkx$, preserving its parent-child structure. The generated CFGs are stored in a list, where each element is a directed CFG (networkx.DiGraph()).

C. Vulnerability Identification Module

This module integrates three functions: the extraction of the vulnerability-critical statement sequence, selection of related commits and files, and matching vulnerability files. Before constructing the data flow graph, the Python library re is used to extract the parameters from the classified statements. We implement our algorithm to generate the data flow graph(DFG) relevant to the parameters, and then the vulnerability-critical statements are extracted from the DFG. To identify related commits and files, we implement a stack-based mechanism that tracks function declarations to detect changes in function declarations. When a modification is found, the original declaration is pushed to the stack. Since commits are analyzed in reverse order, this ensures accurate tracing of the vulnerability's evolution. This implementation of matching with vulnerable files employs a line-by-line and character-by-character matching approach to ensure that the statements in $VCSeq$ appear exactly and in the same order within the target file, except that it ignores minor formatting changes, such as spaces. Regular expressions are used to tolerate non-semantic edits, such as whitespace or comment changes, then the statements from $VCSeq$ and the target file are compared as two character strings.

VI. Evaluation

We conducted the evaluation on a desktop machine equipped with an Intel(R) Core(TM) i7-14700KF running at 3.40GHz, and 64 GB of RAM. We evaluated VicDiff by answering four research questions.

- **RQ1:** What are the statistics of patching patterns? Are the patterns balanced or not?
- **RQ2:** How effective are the key components in VicDiff?
- **RQ3:** How precise is VicDiff on datasets from different projects? How effective is VicDiff compared to state-of-the-art patch-based methods?
- **RQ4:** How efficient is VicDiff in terms of running time?
- **RQ5:** What are the failure modes and potential areas of improvement for VicDiff?

Datasets. We selected the Linux kernel for evaluation for the following reasons: (1) a large number of vulnerabilities with diverse types; (2) frequent and complex code upgrades; and (3) well-maintained and publicly available documentation. To further evaluate the generalizability of our approach, we also randomly constructed a small set of data from other software projects, including OpenSSL, Wget, MySQL, and FFmpeg. We first collected CVEs of the Linux kernel or other projects from the NVD. If the patch URLs of a CVE entry are not found or, for Linux kernel, the patch URLs are distributed on websites other than git.kernel.org/stable/, github.com/torvalds/, git.kernel.org/pub/scm/, and git.kernel.org/cgit/, these CVEs are excluded from our data set. Finally, the data sets consist of (1) 6,920 CVEs and 5,859,238 code commits for the Linux kernel; (2) 6 CVEs and 15 commits for OpenSSL; (3) 3 CVEs and 7 commits for Wget; (4) 10 CVEs and 22 commits for MySQL; and (5) 4 CVEs and 12 commits for FFmpeg. In total, there are 6943 CVEs in the datasets.

Ground truth. In our evaluation, verification of TP, FP, and FN relies on manual labeling of the ground truth because there are no open-source ground truth datasets for our task. As it requires much work for labelling all CVEs, commits, and files in our datasets, we employed probability sampling to ensure representativeness and the reliability of statistical inference through the principle of randomness. After VicDiff processed all CVEs in our datasets, we randomly selected and manually labeled 112 CVEs and their 880 files/commits with ground truth for the Linux dataset. For other datasets with a small number of CVEs, we labeled all records with ground truth. Therefore, we have 135 CVEs and 936 files/commits with ground truth labels. We adopted a systematic verification

approach to ensure the accuracy and credibility of the ground truth by manually inspecting CVE descriptions, changelog, and source code.

A. Patching Patterns(RQ1)

Firstly, we ran VicDiff with all patches in the datasets and classified each patch into one of the three patch categories: Insertion-Only Patches, Deletion-Only Patches, and Insertion and Deletion Patches. The statistical analysis, as shown in Table I, shows that (1) Insertion-Only Patches account for 23.56 %, these patches usually repair vulnerabilities such as buffer overflow or integer overflow by adding check conditions. (2) Deletion-Only Patches account for only 1.82%, because the statements in the patches of this category are unnecessary for the functionality, simply deleting them does not disrupt the original function of the source code. As it is a rare situation in a software project, the percentage of this category is tiny; and (3) Insertion and Deletion Patches dominate with a proportion of 74.62%. This proportion indicates that simply removing erroneous statements or adding new statements is not sufficient to fix vulnerabilities; the majority of patches require both removing existing statements and introducing new statements to achieve the necessary fixes.

Project	Ins	Del	InsDel	Total
Linux-kernel	1635	126	5159	6920
Openssl	0	0	6	6
Wget	1	0	2	3
FFmpeg	0	0	10	10
Mysql	0	0	4	4
Total	1636	126	5181	6943
Proportion	**23.56%**	**1.82%**	**74.62%**	100%

TABLE I
PATCH CLASSIFICATIONS IN OUR DATASET

Then VicDiff classified each statement in a patch. At first, every statement in an Insertion-Only patch is classified as Pattern A, and in a Deletion-Only patch as Pattern R. Then, VicDiff classified each statement in an Insertion and Deletion patch into one of the seven patterns based on differential analysis. The results are shown in Table II and Table III. The reported data are 135 patches and 683 statements from the ground truth datasets. In all these patches, there are 314 statements of Pattern A, and the percentage reaches 45.97%, which is the highest. Pattern M.5 is 15.67%, the highest among Pattern M, and M.4 is 3.95%, the lowest.

The experiments show that the distribution of statements in different patterns is roughly balanced, and every patching pattern is supported by real-world cases.

B. Ablation Study(RQ2)

VicDiff consists of three key components: noise filtering, statement sequence extraction, and matching. To validate the effectiveness of these components, we conducted an ablation study.

Project	A	R	M	Total
Linux-kernel	238	35	243	516
Openssl	13	1	12	26
Wget	19	3	12	34
FFmpeg	7	2	22	31
Mysql	37	16	23	76
Total	314	57	312	683
Proportion	**45.97%**	**8.35%**	**45.68%**	**100%**

TABLE II
STATEMENT PATTERN A-R-M

Project	M.1	M.2	M.3	M.4	M.5
Linux-kernel	46	46	32	23	96
Openssl	8	4	0	0	0
Wget	0	8	0	0	4
FFmpeg	12	6	0	4	0
Mysql	4	12	0	0	7
Total	70	76	32	27	107
Proportion	**10.25%**	**11.13%**	**4.68%**	**3.95%**	**15.67%**

TABLE III
PATTERN M.1-M.5

First, we validated the effectiveness of the noise filter module through ablation experiments. Specifically, we compared two settings: one with the noise filter (referred to as VicDiff) and one without it (referred to as VicDiff-nf). We conducted experiments on 135 CVEs where other modules in VicDiff were unchanged. The experimental results are shown in Figure IV as FN (False Negatives) and FP (False Positives). The results show that VicDiff-nf has 14 more false negatives than VicDiff. For example, in CVE-2023-45863, the function definition was changed from static void *fill_kobj_path(const struct kobject *kobj, char *path, int length)* to static int *fill_kobj_path(const struct kobject *kobj, char *path, int length)*, causing a failure in matching with earlier commits and resulting in FN for VicDiff-nf. As another example, in CVE-2023-6111, the method extraction for the function *nft_setelem_deactivate* was not filtered by VicDiff-nf; therefore, the statement *if (catchall->elem == elem_priv)* was included in the $VCSeq$. This statement was modified earlier than the VIC, causing an increased false negative (FN) rate. Furthermore, our case study confirmed that none of the critical statements were removed by the noise filtering process.

Then we evaluated two critical statement identification approaches: one is the method based differential analysis(referred to as VicDiff), and one is the method using all deletion statements as critical statements(referred to as VicDiff-ds). We keep other components unchanged in this experiment. The final results are shown in the figure as the FP and FN of VicDiff-ds. Experimental results show that compared to VicDiff, VicDiff-ds produces significantly more FPs and FNs. The reason for the poor performance of VicDiff-ds is that many critical statements are not included.

The experimental results demonstrate that noise filtering

is effective in reducing the false negative. The overall improvement in performance is mainly attributed to the approach of vulnerability-critical statements taken by VicDiff.

Project	VicDiff-nf		VicDiff-ds		VicDiff	
	FP	FN	FP	FN	FP	FN
Linux-kernel	39	124	196	314	39	110
Openssl	0	1	5	2	0	1
Wget	0	0	2	0	0	0
FFmpeg	1	3	1	4	1	3
Mysql	2	3	2	4	2	3

TABLE IV
ABLATION OF FILTERING AND REMOVAL OF NOISE STATEMENTS

C. Effectiveness (RQ3)

To evaluate the efficiency of VicDiff, we experimented on the datasets with ground truth labels. To demonstrate the improvement of our approach, we chose three tools, namely B-SZZ [15], V-SZZ, and Redebug for comparison. These three tools are used for the detection of vulnerable source code and are based on patch analysis, like ours. We did not use VIC tools based on symbolic execution or PoC migration because they require PoC as input; however, for CVEs in our datasets, some of the PoCs are unavailable.

Table V presents the precision and recall rates of the four tools. The results show that VicDiff has a precision of 94.94% and a recall of 86.92% for the Linux kernel, significantly outperforming the other three tools. B-SZZ has the lowest precision and recall, and V-SZZ is better than B-SZZ and Redebug. Compared to V-SZZ [16], VicDiff has much lower false positives while false negatives remain low, as VidDiff has a larger set of vulnerability-critical statements.

For example, there are 3 commits in Figure 8 for CVE-2023-6111. COM1 is the security patch, COM2 is the VIC, and COM3 is the last commit to generate a source file without this vulnerability before the VIC. The commits between COM1 and COM2 are not illustrated. Among the four tools, only VicDiff correctly identifies COM2 as VIC.

B-SZZ traced all deletion statements and determined the file versions committed by COM2 and COM3 as bug-inducing changes. V-SZZ is based on the assumption that VIC is the first commit that introduced a deletion statement in the patch; therefore, it identified the files committed by COM3 as the first vulnerable file. Redebug [22] encountered an earlier mismatch between COM1 and COM2, as it is a general-purpose tool that uses exact line matching to locate vulnerabilities, by matching all statements in a patch, including the positioning lines. VicDiff significantly outperforms other baseline methods in OpenSSL and Wget projects, demonstrating generalizability and effectiveness. However, its performance in the FFmpeg and MySQL datasets is slightly lower than that of V-SZZ. As shown in V, VicDiff has only one more false negative than V-SZZ, and one more false positive for MySQL. By investigating these cases, we found that V-SZZ had a smaller set of critical statements; therefore, it was less affected by refactoring.

Tool	TP	FP	FN	Precision	Recall
B-SZZ	357	196	327	64.56%	52.19%
V-SZZ	533	259	108	64.30%	83.15%
Redebug	371	138	156	72.89%	70.40%
VicDiff	731	39	110	**94.94%**	**86.92%**

(a) Linux-kernel Result

Tool	TP	FP	FN	Precision	Recall
B-SZZ	10	2	5	83.33%	66.67%
V-SZZ	9	4	2	69.23%	81.82%
Redebug	7	7	1	50%	87.5%
VicDiff	14	0	1	**100**%	**93.33**%

(b) Openssl result

Tool	**TP**	**FP**	**FN**	**Precision**	**Recall**
B-SZZ	5	2	0	71.43%	100%
V-SZZ	5	2	0	71.43%	100%
Redebug	2	4	1	33.33%	66.67%
VicDiff	7	0	0	**100**%	**100**%

(c) Wget Result

Tool	**TP**	**FP**	**FN**	**Precision**	**Recall**
B-SZZ	17	1	4	94.44%	80.95%
V-SZZ	16	4	2	80%	**88.89**%
Redebug	8	5	9	61.54%	47.06%
VicDiff	18	1	3	**94.74**%	85.71%

(d) FFmpeg Result

Tool	**TP**	**FP**	**FN**	**Precision**	**Recall**
B-SZZ	6	2	4	75%	60%
V-SZZ	9	1	2	**90**%	**81.82**%
Redebug	1	8	3	11.11%	25%
VicDiff	7	2	3	77.78%	70%

(e) Mysql Result

TABLE V
PERFORMANCE OF DIFFERENT APPROACHES

We conducted a performance analysis based on the type of vulnerabilities. The ground truth dataset includes 135 CVEs across 44 types of vulnerabilities. We identified the top five most common vulnerability types in the dataset and reported their performance, as shown in Figure VI.

CWE-362 and CWE-20 achieved the highest precision and recall, mainly because this type of vulnerability is typically patched by adding or removing locking statements, resulting in a relatively small number of statements in the $VCSeq$, which leads to fewer false positives during trace-back.

On the other hand, CWE-476 has the highest number of false negatives (FNs). This is mainly because patches of this type tend to contain a larger number of statements, resulting in a larger $VCSeq$.

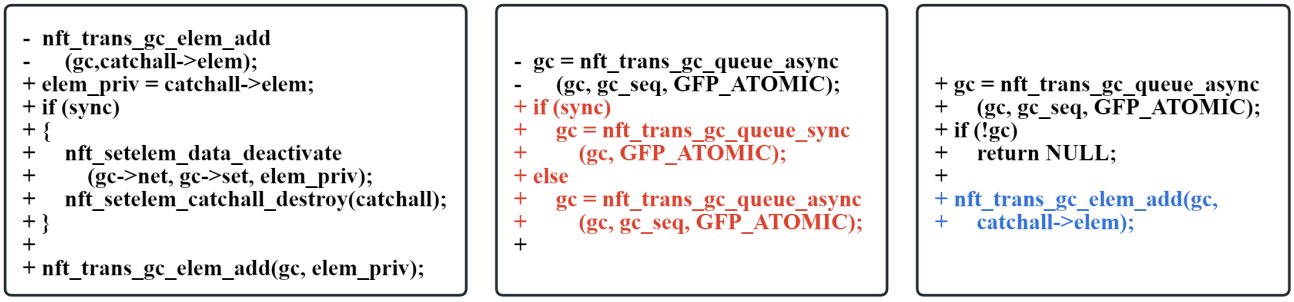

Fig. 8. The commit history of the running example CVE-2023-6111.

CWE ID	CWE Name	CVE	precision	Recall
416	Use After Free	18	87.16%	87.56%
476	NULL Pointer Dereference	18	94.7%	80.65%
362	Concurrent Execution using Shared Resource with Improper Synchronization	6	100%	100%
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	6	100%	86.96%
20	Improper Input Validation	5	100%	100%

TABLE VI
THE PERFORMANCE OF VICDIFF BY TOP 5 CWE

Finally, we compared the results of VicDiff with the Known Affected Software Configurations data provided by NVD on the 6920 CVEs of the Linux kernel project. We found that 3826 CVEs had no corresponding entries in the Known Affected Software Configurations or records of the VICs; 767 VICs from NVD mismatched with VicIff, and only 2327 CVEs matched with VicDiff.

The experimental results show that VicDiff outperforms state-of-the-art tools, especially in reducing false positives.

D. Efficiency (RQ4)

VicDiff is a static analysis tool specially designed to find Vulnerability-Inducing Commits (VIC). We experimented with VicDiff on our datasets and evaluated the efficiency of our approach. To obtain these results, we measured the execution time over all CVEs in the ground truth dataset, running the experiments three times and reporting the average. VicDiff generates a $VCSeq$ for each patch and then matches it with the commit history. The time required for patch analysis to generate the $VCSeq$ is 0.269 seconds. For commit matching, the average time taken to match against 100 commits is 0.255 seconds. Overall, the time for finding a VIC—including both patch analysis and commit history matching—ranges from 0.199 seconds to 2.125 seconds, with an average of 0.889 seconds. By comparison, methods based on symbolic execution are much slower. For example, SymSetion allocated a single CPU core for a maximum of 10,000 seconds for a single symbolic execution [23].

The experiments show that VicDiff is efficient in finding VIC and is scalable up to large-scale projects.

E. Case Study (RQ5)

In our experiment, we have 117 false negatives and 42 false positives for VicDiff. We did a case study and found three reasons attributed to the failure of VicDiff: (1) refactoring of critical statements, (2) including non-critical statements, and (3) failure to include critical statements.

False negative analysis. The leading cause of false negatives (FN) is that some statements in the $VCSeq$ undergo refactoring that does not affect the existence of the vulnerability. For example, CVE-2016-10044 was refactored from *return mount_pseudo(fs_type, "aio:", NULL, &ops, 0xa10a10a1);* to *return mount_pseudo(fs_type, "aio:", NULL, &ops, AIO_RING_MAGIC);*. Although this change does not affect the vulnerability's existence, the mismatch causes VicDiff to terminate too early. Another reason is the inclusion of non-critical statements into $VCSeq$, whose subsequent refactoring also leads to mismatches. For instance, in CVE-2023-3776, the $VCSeq$ includes the non-critical statement *else if (head->mask != 0xFFFFFFFF)*, whose modification is irrelevant to the vulnerability but causes a match failure.

False positive analysis. The main reason for FP is the failure to include critical statements. For example, CVE-2023-3212 is a Use-After-Free vulnerability. Our tool automatically extracted a vulnerability-critical statement sequence that includes three statements. However, another critical statement *ret = gfs2_trans_begin(sdp, 0, sdp->sd_jdesc->jd_blocks)* is located in a function other than the one in the patch. Therefore, it was not included in the extracted sequence by VicDiff.

The findings suggest that VicDiff can be improved by better matching strategy for refactoring and more accurate identification of critical statements.

VII. Discussion

A. Scope and Limitations

The datasets in our evaluation contain only C/C++ projects. Because the syntax, semantics, and types of vulnerabilities of one language are different from those of other languages, our approach cannot be used directly on other languages. Nevertheless, it is promising to extend to other languages such as Java and Python with specific adaptations. Our approach is limited to single-file patches and in-function analysis due to challenges such as state explosion when constructing vulnerability-critical statement sequences for multi-file patches and cross-function analysis. We argue that while most statements on the path to the vulnerability triggering point are essential, when a new vulnerability is introduced, during version upgrades, the most critical statements lie around the patch statements, so it is effective to consider only in-function analysis.

B. Applicability

Our approach produces vulnerability-critical statement sequences with minimum length. Therefore, even if earlier versions of the vulnerable file underwent radical code changes, the adverse effect will be minimized. The availability of vulnerability patches is required. Although some patches are hard to find, in most cases, the patches are available from the official websites or can be identified in the commit history. Another concern is the incorrectness of patches. The errors in the patches primarily concern the correction logic for the vulnerabilities, such as the insert statements in the patch. However, when the variables associated with the vulnerability are correctly identified, the performance of our algorithm will not be adversely affected. For any new CVEs, our approach is applicable as long as patches are available.

VIII. Related Work

Patch based detection for VIC Sliwerski et al. [15]first developed the method to detect bug introduction changes, which was later named B-SZZ (basic-SZZ) and improved as AG-SZZ [24], DJ-SZZ [25], MA-SZZ [26], and L&R-SZZ [27]. However, these methods primarily focus on deletion statements in patches but fail to consider other statements that contribute to the bug. V-SZZ [16] is based on the assumption that vulnerabilities are introduced in the initial version that are related to the deletion statements. This method significantly improved the recall rate compared to other methods. However, its assumption is not always valid, and its failure to account for renaming behaviors in historical versions limits its improvement in precision and recall. Shi et al. [17] employed a method based on manually defined vulnerability functions to accurately identify vulnerabilities-affected versions. However, it has a heavy dependence on manually defined rules, limiting its adaptability in large-scale data sets. He et al. [28] utilized information from developers' changelogs to improve the precision. However, they applied matching of deletion statements to detect the initial vulnerable version in earlier versions, such as SZZ. Our approach is also patch-based and outperforms other work in the precision of vulnerability logic by patching patterns analysis.

Triggering analysis based detection for VIC If a vulnerability can be triggered with a seed input, it can safely claim that the software version under test is vulnerable. Fuzzing technique is generally used for finding unknown vulnerabilities. Directed fuzzing can also be used to verify that an earlier version of software is vulnerable to a known CVE [29], [30], [31], [32]. However, it is usually tough to reach and trigger a vulnerability. It takes time, especially for vulnerabilities with deep paths and complicated triggering conditions. The execution traces of Proof of Concept (PoC) are used to guide seed generation [33] or symbolic execution [23] to verify if earlier versions are vulnerable; however, PoC for a CVE is not commonly available.

AI assisted detection for VIC Machine learning can be used to train models on vulnerable source files and other vulnerability reports to detect known and unknown vulnerabilities [34], [35], [36], [37]. However, these methods face a significant challenge: the changes between earlier vulnerability-related commits are often subtle, making it difficult for machine learning algorithms to distinguish these variations. Risse et al. [38] conducted research and experiments to confirm the existence of this problem. Large language models (LLMs) [39], [40], [41], [42] are applied not only in natural language processing but also in programming language and vulnerability detection. Although they are promising in assisting the understanding of vulnerability reports and vulnerability logics of patches, it was demonstrated that the state-of-the-art LLMs are challenging to capture the nuanced semantics among patched and unpatched file versions [43].

Known vulnerability detection Vulnerability introducing commit identification is a specific case of the known vulnerability detection. Open source tools such as SonarQube and CppCheck are used for general-purpose security review of source code and are not capable of accurately identifying VIC. Clone detection [44], [45], [46] is an important technique to detect known vulnerabilities in projects with source code reuse. The method is based on comparing the vulnerable files to the files from other projects that possibly reuse the known vulnerable components. This method is based on the assumption that if a file is similar to the vulnerable file, then it is also vulnerable. However, this approach is not suitable for identifying VIC, as files from earlier versions are primarily similar to the vulnerable file. Patch presence detection [47], [48] is based on comparing the patch with files. If the patch is present in a file similar to the vulnerable file, the file is fixed to prevent vulnerability. However, the method cannot be used for VIC identification, because most patches are developed for the latest version; the patch for the earlier version simply does not exist.

IX. Conclusion

Identifying VIC is an important and challenging problem. To utilize the rich semantics in the patches, we propose the concept of patching pattern and the technique of differential

analysis. By focusing on the differences in statements' location, operator, and parameters in the control of flows of vulnerable and patched files, we can capture the nuances of patching semantics and generate the vulnerability-critical statement sequence. Our method is a lightweight static analysis of the patches and commits, making it efficient and scalable. The experimental results demonstrate that the proposed method significantly outperforms the existing approaches in accuracy.

REFERENCES

[1] Black Duck Software, "Black duck software," https://www.blackduck.com/, 2024, accessed: 2025-01-21.

[2] N. vulnerability database, https://nvd.nist.gov/, 2024.

[3] O. S. V. Database, http://www.osvdb.org, 2024.

[4] Bugtraq, http://www.securityfocus.com, 2024.

[5] "Hakiri: Ships secure ruby apps," https://hakiri.io/, 2019, [Online; accessed 2025-01-07].

[6] "Snyk: Develop fast: Stay secure," https://snyk.io/, 2019, [Online; accessed 2025-01-07].

[7] "Sourceclear: Software composition analysis for devsecops," https://www.sourceclear.com/, 2019, [Online; accessed 2025-01-07].

[8] "Bundler-audit," https://github.com/rubysec/bundler-audit, 2019, [Online; accessed 2025-01-07].

[9] "Sonatype — oss index," https://ossindex.sonatype.org/, 2019, [Online; accessed 2025-01-07].

[10] "Owasp dependency check," https://www.owasp.org/index.php/OWASP_Dependency_Check, 2019, [Online; accessed 2025-01-07].

[11] V. H. Nguyen and F. Massacci, "The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 493–498.

[12] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 869–885.

[13] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 919–936.

[14] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen, "Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 4255–4269, 2021.

[15] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[16] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2352–2364.

[17] Y. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang, "Precise (un) affected version analysis for web vulnerabilities," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[18] CVE, "Common Vulnerabilities and Exposures (CVE)," https://cve.org, 2025, accessed: 2025-01-12.

[19] MITRE Corporation, "MITRE Corporation: Driving Innovative Solutions," https://www.mitre.org, 2025, accessed: 2025-01-12.

[20] L. Torvalds and J. Hamano, *Git Manual and Source Code Documentation*, Git Project, 2005.

[21] National Institute of Standards and Technology (NIST). (2023) CVE-2023-6111: Linux Kernel Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2023-6111.

[22] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 48–62.

[23] Z. Zhang, Y. Hao, W. Chen, X. Zou, X. Li, H. Li, Y. Zhai, and B. Lau, "Symbisect: Accurate bisection for fuzzer-exposed vulnerabilities," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2493–2510.

[24] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[25] C. Williams and J. Spacco, "Szz revisited: verifying when changes induce fixes," in *Proceedings of the 2008 workshop on Defects in large software systems*, 2008, pp. 32–36.

[26] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[27] S. Davies, M. Roper, and M. Wood, "Comparing text-based and dependence-based approaches for determining the origins of bugs," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 107–139, 2014.

[28] Y. He, Y. Wang, S. Zhu, W. Wang, Y. Zhang, Q. Li, and A. Yu, "Automatically identifying cve affected versions with patches and developer logs," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 2, pp. 905–919, 2023.

[29] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," 2017.

[30] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[31] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors," *arXiv preprint arXiv:2402.03704*, 2024.

[32] C. Chen, V. Gohil, R. Kande, A.-R. Sadeghi, and J. Rajendran, "Psofuzz: Fuzzing processors with particle swarm optimization," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[33] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating vulnerability assessment through poc migration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3300–3317.

[34] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM computing surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[35] U. Sarmah, D. Bhattacharyya, and J. K. Kalita, "A survey of detection methods for xss attacks," *Journal of Network and Computer Applications*, vol. 118, pp. 113–143, 2018.

[36] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "Vulchecker: Graph-based vulnerability localization in source code," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6557–6574.

[37] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "Ivysyn: Automated vulnerability discovery in deep learning frameworks," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2383–2400.

[38] N. Risse and M. Böhme, "Uncovering the limits of machine learning for automatic vulnerability detection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4247–4264.

[39] Y. Liu, L. Gao, M. Yang, Y. Xie, P. Chen, X. Zhang, and W. Chen, "Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models," *arXiv preprint arXiv:2406.07595*, 2024.

[40] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 47–51.

[41] Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang, "How far have we gone in vulnerability detection using large language models," *arXiv preprint arXiv:2311.12420*, 2023.

[42] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2023, pp. 112–119.

[43] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *2024 IEEE symposium on security and privacy (SP)*. IEEE, 2024, pp. 862–880.

[44] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "V0finder: Discovering the correct origin of publicly reported software vulnerabilities," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3041–3058.
[45] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.
[46] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd annual conference on computer security applications*, 2016, pp. 201–213.
[47] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "Pdiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1149–1163.
[48] J. Dai, Y. Zhang, Z. Jiang, Y. Zhou, J. Chen, X. Xing, X. Zhang, X. Tan, M. Yang, and Z. Yang, "Bscout: Direct whole patch presence test for java executables," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1147–1164.
[49] CVE-2023-40791: Linux Kernel Vulnerability. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-40791
[50] CVE-2023-1476: Linux Kernel Vulnerability. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-1476
[51] CVE-2023-45871: Linux Kernel Vulnerability. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-45871
[52] CVE-2023-46862: Linux Kernel Vulnerability. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-46862
[53] National Institute of Standards and Technology (NIST). (2023) CVE-2023-45863: Linux Kernel Vulnerability. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-45863

APPENDIX A
MOTIVATING EXAMPLES FOR M PATTERNS

Pattern M.1 CVE-2023-40791 [49] is a vulnerability of memory leak or reference count overflow. As shown in Figure 9, this vulnerability involves an error in *extract_user_to_sg()* function in the Linux kernel, which releases a user space page that is still pinned. The patch replaces *put_page()* (line 4) with *unpin_user_page()* (line 5), to unpin the page and decrease the reference count, which fixes the problem. By removing line 4 and adding line 5, the deletion statement in CFG of F_v and the insertion statement in CFG of F_p are in the same position of the same function $extract\_user\_to\_sg$, and the positioning nodes, including line 3 (*while*) and line 6 (*return*) in the two CFGs, remain unchanged, so that the control flows through this pair of statements in F_v and F_p are identical. According to the rule of fully identical positioning nodes, this pair of deletion and insertion statements is classified as Pattern M.1. According to the vulnerability-critical statement sequence extraction rule in Pattern M.1, line 4 in $VCSeq$.

```
1 @@ -1148,7 +1148,7 @@ static ssize_t extract_user_to_sg(struct iov_iter *iter,
2     failed:
3       while (sgtable->nents ¿ sgtable->orig_nents)
4 -       put_page(sg_page(&sgtable->sgl[-sgtable->nents]));
5 +       unpin_user_page(sg_page(&sgtable->sgl[-sgtable->nents]));
6       return res;
```

Fig. 9. CVE-2023-40791.

Pattern M.2 CVE-2023-1476 [50] is a use-after-free vulnerability. As shown in Figure 10, when a thread is in a *mremap()* system call, the *move_page_tables()* function is used to migrate page table entries from an old virtual address range to a new one. If another thread (e.g., *try_to_unmap_one()*) simultaneously performs reverse mapping (*rmap*) operations and accesses the old page address, a race condition may occur. If the rmap lock is not held by the first thread, the old physical page (*old_pfn*) is freed while stale TLB entries still reference it. User-space accesses to *new_addr* may still point to the freed *old_pfn*, resulting in illegal memory access or a use-after-free vulnerability. The patch addresses this issue by replacing *need_rmap_locks()* with a forced true value in two calls to the *move_pgt_entry()* function (by removing lines 3 and 8, and adding lines 4 and 9). This modification ensures that the rmap lock is always acquired when performing page table migrations at the PMD or PUD level.

By removing line 3 and adding line 4, the deletion statement in the CFG of F_v and the insertion statement in the CFG of F_p appear in the same position within the function *move_page_tables*, as their previous and subsequent positioning nodes, such as line 2 and line 5, remain unchanged. This ensures that the control flows through this pair of statements in F_v and F_p have identical previous and subsequent positioning nodes. According to the rule of identical previous and subsequent positioning nodes, this pair of deletion and insertion statements is classified as Pattern M.2. The same applies to line 8 and line 9.

Note that lines 2–3 and 7–8 are split across two lines due to the length of the statements. For analysis purposes, we treat each pair as a single statement. According to the extraction rule defined for vulnerability-critical statement sequences in Pattern M.2, we directly trace back to lines 2, 3, 7, and 8 in earlier commits.

```
1 @@ -504,7 +504,7 @@ unsigned long move_page_tables(struct vm_area_struct *vma,
2     if (move_pgt_entry(NORMAL_PUD, vma, old_addr, new_addr,
3 -       old_pud, new_pud, need_rmap_locks))
4 +       old_pud, new_pud, true))
5       continue;
6 @@ -504,7 +504,7 @@ unsigned long move_page_tables(struct vm_area_struct *vma, gfp_mask)
7     if (move_pgt_entry(NORMAL_PUD, vma, old_addr, new_addr,
8 -       old_pud, new_pud, need_rmap_locks))
9 +       old_pud, new_pud, true))
10      continue;
```

Fig. 10. CVE-2023-1476.

Pattern M.3 CVE-2023-45871 [51] is a vulnerability of incorrect calculation of buffer size. As shown in Figure 11, the *igb_set_rx_buffer_len()* function is responsible for configuring the size of each receive ring buffer. If the buffer is too small, it may not be able to accommodate the actual incoming network frames, under one of two conditions: (1) the frame size exceeds the maximum value allowed by the default skb construction (*IGB_MAX_FRAME_BUILD_SKB*); or (2) the hardware enables the Store Bad Packet (SBP) flag, allowing the reception of erroneous frames for analysis or logging.

In the earlier version, in line 12, only the first condition was checked. However, this ignored the case where the SBP flag is set. In that situation, the receive buffer must also be large enough to hold ”bad frames”; otherwise, it may lead to memory overwrite or packet loss. Lines 12, 13, 16, and 17 collectively introduce a new condition in the patch: if the *E1000_RCTL_SBP* (Store Bad Packet) flag is set (line 17), the large buffer is forcibly enabled. This ensures that, even if the frame size does not exceed the skb construction limit, sufficient buffer space is still allocated when SBP is enabled.

By removing line 12 and adding line 16, the deletion statement in the CFG of F_v and the insertion statement in the CFG of F_p appear in the same function *extract_user_to_sg*, with identical previous or subsequent positioning nodes—for example, line 11 remains unchanged in both CFGs. Moreover, both line 12 and line 16 are if statements, indicating the same type of control structure. According to the rule of identical previous or subsequent positioning nodes with the same control statement type, this pair of deletion and insertion statements is classified as Pattern M.3.

```
1 @@ -4814,6 +4814,10 @@ void igb_configure_rx_ring(struct igb_adapter
  *adapter,
2    static void igb_set_rx_buffer_len(struct igb_adapter *adapter,
3         struct igb_ring *rx_ring)
4    {
5  + #if (PAGE_SIZE <8192)
6  + struct e1000_hw *hw = &adapter->hw;
7  + #endif
8         /* set build_skb and buffer size flags */
9         clear_ring_build_skb_enabled(rx_ring);
10 @@ -4824,10 +4828,9 @@ static void igb_set_rx_buffer_len(struct
  igb_adapter *adapter,
11   #if (PAGE_SIZE <8192)
12 -       if (adapter->max_frame_size <= IGB_MAX...)
13 -           return;
14 -
15 -       set_ring_uses_large_buffer(rx_ring);
16 +       if (adapter->max_frame_size >IGB_MAX... ||
17 +       rd32(E1000_RCTL) & E1000_RCTL_SBP)
18 +           set_ring_uses_large_buffer(rx_ring);
19       #endif
20       }
```

Fig. 11. CVE-2023-45871.

Pattern M.4 CVE-2023-46862 [52] is a vulnerability of null pointer dereference. As shown in Figure 12, the vulnerability is caused by a race condition: when io_uring_show_fdinfo() attempts to access the pid and cpu information of the *sq->thread*, another thread may concurrently free the data structure associated with the SQ (Submission Queue) thread. This can result in *sq->thread == NULL*, and any further access to *sq->thread* will trigger a NULL pointer dereference. Although lines 17 and 18 include a *sq* conditional check, it is not sufficient. If the *sq->thread* is freed immediately after the check (due to the absence of proper locking), it can still lead to a NULL pointer dereference. To address this, lines 8 and 13 introduce a lock acquisition (*sq->lock*) before accessing *sq->thread*, ensuring the pointer remains valid during use.

In the patch, lines 17–20 fall under Pattern M.4 (Identical vulnerable operators). The data flow context statements of the parameter *sq* flowing through line 17 (*seq_printf(m, ”SqThread:\t%d\n”, sq ? task_pid_nr(sq->thread): -1);*) and line 18 (*seq_printf(m, ”SqThreadCpu:\t%d\n”, sq ? task_cpu(sq->thread): -1);*) are line 3 (*sq = ctx->sq_data;*) and line 5 (*sq = NULL;*), respectively.

```
1 @@ -143,13 +143,19 @@ __cold void io_uring_show_fdinfo(struct
  seq_file *m, struct file *f)
2    if (has_lock && (ctx->flags & IORING_SETUP_SQPOLL)) {
3  -     sq = ctx- >sq_data;
4  -     if (!sq->thread)
5  -         sq = NULL;
6  +     struct io_sq_data sq = ctx- >sq_data;
7  +
8  +     if (mutex_trylock(&sq->lock)) {
9  +         if (sq->thread) {
10 +             sq_pid = task_pid_nr(sq->thread);
11 +             sq_cpu = task_cpu(sq->thread);
12 +         }
13 +         mutex_unlock(&sq->lock);
14 +     }
15   }
16
17 -   seq_printf(m, ”SqThread:\t%d\n”, sq ? ...(sq->thread)...);
18 -   seq_printf(m, ”SqThreadCpu:\t%d\n”, sq ? ...(sq->thread)...);
19 +   seq_printf(m, ”SqThread:\t%d\n”, sq_pid);
20 +   seq_printf(m, ”SqThreadCpu:\t%d\n”, sq_cpu);
21     seq_printf(m, ”UserFiles:\t%u\n”, ctx->nr_user_files);
```

Fig. 12. CVE-2023-46862.

Pattern M.5 The patch is the same as Pattern M.3 of CVE-2023-45871, shown in Figure 11. Lines 15 and 18 represent the setting of a flag in the *rx_ring* object, indicating that the ring should use large-page memory allocation. These match the criteria of Pattern M.5 (Identical vulnerable statements) and are thus classified under Pattern M.5. The control predecessor and successor nodes of line 18(*set_ring_uses_large_buffer(rx_ring);*) are line 16 and line 17(*if (adapter->max_frame_size >IGB_MAX_FRAME_BUILD_SKB ||rd32 (E1000_RCTL) & E1000_RCTL_SBP)*)

APPENDIX B
DETAILS OF THE CASE STUDY

We present two case studies in detail, which highlight the failure modes and potential areas of improvement for VicDiff.

False negative analysis. For example, in CVE-2023-45863(Figure 13) [53], which is an Out-of-Bounds Write vulnerability, the root cause lies in the lack of validation after subtracting cur from length. If length becomes negative or

zero, subsequent operations such as memcpy or *(path + –length) may result in out-of-bounds writes. For this vulnerability, our tool extracted a vulnerability-critical sequence that includes lines 2, 5, 15, 18, and 23. During the process of tracing back to the VIC, the original line 15 in the code *path = kmalloc(len, gfp_mask);* was refactored to *path = kzalloc(len, gfp_mask);*, a change that preserves semantics, but alters syntax. Because this change caused the vulnerability-critical statement sequence to no longer match exactly, the tool incorrectly concluded that earlier versions did not contain the vulnerability, resulting in a false negative.

```
1 @@ -121,12 +121,16 @@ static void fill_kobj_path(const struct kobject *kobj, char *path, int length)
2       length -= cur;
3 +     if (length <= 0)
4 +         return -EINVAL;
5       memcpy(path + length, kobject_name(parent), cur);
6       *(path + –length) = '/';}
7     pr_debug("kobject: '%s' (%p): %s: path = '%s'\n", kobject_name(kobj),kobj, __func__, path);
8 +
9 +   return 0;
10 @@ -141,13 +145,17 @@ char *kobject_get_path(const struct kobject *kobj, gfp_t gfp_mask)
11 +retry:
12      len = get_kobj_path_length(kobj);
13      if (len == 0)
14              return NULL;
15      path = kzalloc(len, gfp_mask);
16      if (!path)
17              return NULL;
18 -    fill_kobj_path(kobj, path, len);
19 +    if (fill_kobj_path(kobj, path, len)) {
20 +        kfree(path);
21 +        goto retry;
22 +   }
23      return path;
```

Fig. 13. A Example of False negative.

False positive analysis. Since we adopt an automated approach to extract the vulnerability-critical statement sequence within the vulnerable file and vulnerable function, the complete vulnerability-triggering path is not fully covered. As a result, in cases involving multi-level function calls, some critical statements may reside outside the vulnerable function or even outside the vulnerable file. These critical statements are not included in the extracted sequence. However, if such statements are modified in earlier commits, they may still have a substantial impact on the triggering of the vulnerability, potentially leading to false positive results.

Taking CVE-2023-3212(Figure 14) as an example, this vulnerability is categorized as a Use-After-Free issue. When the function *init_journal()* fails, the program follows the *fail_jindex* path and executes *gfs2_jindex_free()*, which frees the pointer *sdp->sd_jdesc* and sets it to NULL. Immediately afterward, the program calls *iput(sdp->sd_jindex)*, triggering the evict() operation, which eventually leads to the *gfs2_evict_inode()* function. However, *gfs2_evict_inode()* does not check whether *sdp->sd_jdesc* is NULL before invoking the following line in the *evict_linked_inode* function: *ret = gfs2_trans_begin(sdp, 0, sdp->sd_jdesc->jd_blocks);*. This results in a dereference of a previously freed pointer, leading to a Use-After-Free or NULL pointer dereference vulnerability.

Our tool automatically extracted a vulnerability-critical statement sequence that includes lines 3, 21, and 23. However, the statement that ultimately triggers the vulnerability *ret = gfs2_trans_begin(sdp, 0, sdp->sd_jdesc->jd_blocks);* in the *evict_linked_inode* function—was not included in the extracted sequence. As a result, during the process of tracing back to the VIC, the modification to this critical statement occurred earlier than the changes in the identified vulnerability-critical sequence. Consequently, the tool mistakenly concluded that the vulnerability was still present in earlier versions, leading to a false positive result.

```
1 @@ -1419,6 +1419,14 @@ static void gfs2_evict_inode(struct inode *inode)
2   struct super_block *sb = inode->i_sb;
3   struct gfs2_sbd *sdp = sb->s_fs_info;
4   struct gfs2_inode *ip = GFS2_I(inode);
5   struct gfs2_holder gh;
6   int ret;
7   if (inode->i_nlink —— sb_rdonly(sb) —— !ip->i_no_addr)
8     goto out;
9 + /*
10 + *In case of an incomplete mount, gfs2_evict_inode() may be called
11 + *for system files without having an active journal to write to.
12 + *In that case, skip the filesystem evict.
13 + */
14 + if (!sdp->sd_jdesc)
15 +   goto out;
16  gfs2_holder_mark_uninitialized(&gh);
17  ret = evict_should_delete(inode, &gh);
18  if (ret == SHOULD_DEFER_EVICTION)
19    goto out;
20  if (ret == SHOULD_DELETE_DINODE)
21    ret = evict_unlinked_inode(inode);
22  else
23    ret = evict_linked_inode(inode);
```

Fig. 14. A Example of False positive.

APPENDIX C
ARTIFACT APPENDIX

A. Description & Requirements

How to access

The complete artifact is available on GitHub at the following URL: https://github.com/x-s-g/Vuln-Intro. The DOI link is https://doi.org/10.5281/zenodo.17064690.

Hardware dependencies

- Memory: At least 32GB RAM (e.g., we used systems equipped with 64GB RAM in our evaluation).
- CPU: Commodity CPUs (e.g., we used Intel(R) Core(TM) i7-14700KF or equivalent models in our evaluation).

Software dependencies

- python 3.13
- pycparser 2.22
- networkx 3.5
- requests 2.32.3
- lxml 5.3.2

B. Benchmarks

Since the complete dataset used in our evaluation is large, this artifact only includes a sample dataset for demonstration purposes. The complete set of CVE entries and associated data used in our evaluation can be accessed through the links provided via DOI and GitHub.

C. Artifact Installation & Configuration

To prepare the environment for the evaluation, it is recommended to create a conda environment and install all software dependencies listed in the README file. Our release requires no installation. Detailed instructions on using and running the tool are provided in the README.

D. Experiment Workflow

Our workflow consists of four main components:

Data Collection: The first step of our work is designed to support the automated collection of vulnerability-related datasets.

Data Crawling: This phase involves processing the collected data, including preprocessing and filtering out noise statements.

Static Analysis: Perform differential analysis to extract patterns corresponding to code statements.

Identification: Extract vulnerability-critical statement sequences and match them against candidate files to obtain the final results.

E. Major Claims

(C1): The patterns we summarized have been validated on real-world datasets. This conclusion is supported by Experiment (E1), and the corresponding results are presented in Tables II and III of the paper.

(C2): The system outperforms other approaches in terms of accuracy on the vulnerability-introduction point detection task. This conclusion is also verified by Experiment (E1), with detailed results shown in Table V.

F. Evaluation

To evaluate our work, several steps are required, including dataset downloading and model setup. We have included the data collection module and experiment execution module in the artifact package to facilitate quicker reproduction of the experimental results.

Experiment (E1) [72 human-hours + 3 compute-hours]: This experiment performs vulnerability-introducing commit detection on a source code dataset. Although we manually verified only a subset of the data, the samples were randomly selected and consist of real-world vulnerabilities, which is sufficient to support our conclusions. Our evaluation metrics include precision and recall.

[Preparation] After setting up the environment, the user can utilize the scripts in the $Dataset\_Dataset\_getCVElink$ directory to collect all CVE entries along with their corresponding detailed information links. The Dataset directory contains the code and examples for dataset collection. The complete list of CVE entries and dataset links can be found in $CVE\_list.txt$.

In this project, we have only uploaded a subset of runnable examples. The remaining examples can be automatically downloaded and constructed using the provided crawler scripts, $load\_file.py$ and $patch\_list.py$. The full download and processing of all examples is estimated to require approximately 72 compute-hours.

For each CVE, the ground truth data is located in the $groundtruth.txt$ file within the corresponding CVE directory. The experimental output results are stored in the $CVE/result$ directory, with each file named after the corresponding commit ID.

[Execution] The $Data\_Crawling$ directory is used for data preprocessing and filtering noise statements to obtain vulnerability-related statements. Specifically, $re_refactor.py$ and $filter.py$ output the noise statements that have been filtered out.

The $Static\_Analysis$ directory is responsible for differential analysis and modeling of vulnerable functions by constructing their ASTs and CFGs.

In particular, $af\_cfg.py$ and $bf\_cfg.py$ are used to generate the control flow graphs (CFGs) of the vulnerable functions.

The Identification directory is used to extract vulnerability-critical statement sequences and perform matching with earlier commits and corresponding files. Patterns corresponding to vulnerability-related statements can be obtained by running $patch\_label.py$, while the evolution of vulnerable function names can be traced by running $rename\_lower\_version.py$. Finally, the overall matching results can be obtained by executing $match.py$.

[Results] The contents of the $groundtruth.txt$ file need to be manually analyzed to identify the actual commit that introduced the vulnerability. Based on this analysis, the experimental output should be evaluated accordingly. The results are consistent with those presented in Table V.