Silent Taint-Style Vulnerability Fixes Identification

Zhongzhen Wen
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
wenzhongzhen@smail.nju.edu.cn

Jiayuan Zhou
Centre for Software Excellence, Huawei
Waterloo, Canada
jiayuan.zhou1@huawei.com

Minxue Pan*
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
mxp@nju.edu.cn

Shaohua Wang
Central University of Finance and Economics
Beijing, China
davidshwang@ieee.org

Xing Hu
Zhejiang University
Ningbo, China
xinghu@zju.edu.cn

Tongtong Xu
Software Engineering Application Technology Lab, Huawei
Hangzhou, China
xutongtong9@huawei.com

Tian Zhang*
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
ztluck@nju.edu.cn

Xuandong Li
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
lxd@nju.edu.cn

Abstract

The coordinated vulnerability disclosure model, widely adopted in open-source software (OSS) organizations, recommends the silent resolution of vulnerabilities without revealing vulnerability information until their public disclosure. However, the inherently public nature of OSS development leads to security fixes becoming publicly available in repositories weeks before the official disclosure of vulnerabilities. This time gap poses a significant security risk to OSS users, as attackers could discover the fix and exploit vulnerabilities before disclosure. Thus, there is a critical need for OSS users to sense fixes as early as possible to address the vulnerability before any exploitation occurs.

In response to this challenge, we introduce EarlyVulnFix, a novel approach designed to identify silent fixes for taint-style vulnerabilities—a persistent class of security weaknesses where attacker-controlled input reaches sensitive operations (sink) without proper sanitization. Leveraging data flow and dependency analysis, our tool distinguishes two types of connections between newly introduced code and sinks, tailored for two common fix scenarios. Our evaluation demonstrates that EarlyVulnFix surpasses state-of-the-art baselines by a substantial margin in terms of F1 score. Furthermore, when applied to the 700 latest commits across seven projects, EarlyVulnFix detected three security fixes before their respective security releases, highlighting its effectiveness in identifying unreported vulnerability fixes in the wild.

*Corresponding authors.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652139

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Vulnerability, Program Analysis, Software Security

ACM Reference Format:
Zhongzhen Wen, Jiayuan Zhou, Minxue Pan, Shaohua Wang, Xing Hu, Tongtong Xu, Tian Zhang, and Xuandong Li. 2024. Silent Taint-Style Vulnerability Fixes Identification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652139

1 Introduction

In the contemporary software landscape, the increasing complexity of applications has driven a heightened reliance on open-source third-party libraries. Although these libraries offer substantial benefits, including reduced development time and costs, they also introduce security risks for users of open-source software (OSS). Staying vigilant and promptly updating libraries is crucial to mitigate these security threats.

A significant category of vulnerabilities in open-source third-party libraries is taint-style vulnerabilities. In these cases, an attacker-controlled input (source) is propagated to sensitive operations (sink) without proper sanitization. Taint-style vulnerabilities are recognized for their prevalence and severity, with all top 5 vulnerabilities belonging to this category in the top 25 CWEs list [2]. An illustrative example is the "Log4Shell" vulnerability in the widely used Apache Log4j2 library, affecting numerous systems and applications. Mitigating the impact of such vulnerabilities requires collaborative efforts from OSS users, library vendors, and vulnerability finders.

To ensure responsible and coordinated handling of vulnerabilities, the practice of Coordinated Vulnerability Disclosure (CVD) has gained widespread adoption in many OSS organizations [1, 3, 4].

CVD involves a structured process where individuals or organizations discover and privately report vulnerabilities to the responsible entity, allowing them to develop and deploy security patches before public disclosure. This enables OSS users to promptly initiate the remediation process, as patches become accessible upon vulnerability disclosure. To minimize the extent of information dissemination regarding the vulnerability, CVD also recommends addressing vulnerabilities in a **silent** manner. For instance, the commit message should refrain from conveying any details that might expose the nature of the vulnerability. It is expected that these collective efforts will decrease the likelihood of attacks from malicious users.

However, security threats still persist. Various factors, such as limited human resources and extended release cycles, contribute to the variable time span between a security fix and the publication of the release, ranging from days to months [19, 22, 39]. For instance, the Log4Shell vulnerability, considered one of the most significant [12], was disclosed on December 10, 2021, while the corresponding fix had been publicly available 11 days earlier, on November 29, 2021. Once committed to the repository, the security fixes become publicly available given the public nature of OSS development. Security fixes can inadvertently expose valuable information about the vulnerabilities themselves, potentially enabling attackers to exploit vulnerabilities manually or automatically [7, 34]. Consequently, the time gap between fixing vulnerabilities and public disclosure can expose OSS users to significant security risks.

To mitigate these security risks, approaches for identifying silent fixes have been proposed in recent years with the aim of sensing vulnerabilities as early as possible in the development process. Deep learning-based approaches[38, 39] detect silent vulnerability fixes by learning the semantic meaning of code change. These methods involve encoding commits into embedding vectors and subsequently generating probability scores indicating the likelihood of a commit being a security fix. Though these approaches produce promising results, they have limitations that hinder practical use. Challenges include difficulty in collecting data on vulnerability fixes, leading to a generalization problem and suboptimal performance. The extremely imbalanced class distribution of fix data poses challenges for model training, and providing only probabilities of a commit being a vulnerability fix offers limited assistance to human analysts. Moreover, when it comes to identifying taint-style vulnerability fixes, deep learning-based approaches encounter more severe challenges. Taint-style vulnerabilities often span multiple functions or projects, and existing methods capture only the context surrounding code changes, lacking sufficient awareness to effectively learn features associated with fixes for such vulnerabilities.

DAA [10] employs off-the-shelf static analysis security testing (SAST) tools to identify vulnerability fixes in software projects. The approach involves scanning consecutive versions and assessing the removal of security alerts. To identify a vulnerability fix, DAA necessitates two conditions: 1) the existence of a pre-fix alert generated by SAST tools, and 2) the subsequent disappearance of the alert after the fix is introduced. This mandates high recall (for condition 1) and precision (for condition 2) in the underlying SAST tools. However, existing SAST tools struggle to detect real-world vulnerabilities, as highlighted in recent studies [24, 25].

To address the aforementioned challenges, we present EarlyVulnFix, an approach that utilizes program analysis techniques to automatically identify silent taint-style vulnerability fixes. Based on our manual inspection of hundreds of fixes in existing datasets [24, 39]. we observed that nearly all taint-style vulnerability fixes can be categorized into two types. One type, called sanity check fixes, involves directly checking or sanitizing untrusted data to prevent its flow into sinks. This concept aligns with sanitizers in taint analysis [20, 29] and serves as the default fix pattern for taint-style vulnerabilities in related works such as program repair [26, 35] and pattern mining [32, 33]. The other type, permission list fixes, entails creating or appending permission lists to indirectly control the flow of data. Despite being common in real-world fixes, permission list fixes are often overlooked in existing research on taint-style vulnerability. The core concept guiding our approach is the observations that in both fix types, the newly introduced code has connections with sinks, as discussed in Section 2.2. Our approach considers two types of connections between newly introduced code and sinks, tailored for the two types of fixes. Specifically, for sanity check fixes, EarlyVulnFix first identifies "juncture" statements— a specialized type of statement that connects newly introduced code and sinks. Subsequently, EarlyVulnFix constructs a data flow graph (DFG) from juncture methods and detects the data flow from newly introduced code to sinks by traversing the DFG. For permission list fixes, EarlyVulnFix analyzes data dependency, intraprocedural control dependency, and interprocedural embedded-halt control dependency to connect newly introduced code and sinks.

We have implemented our approach as a tool, also named EarlyVulnFix, designed to detect silent taint-style vulnerability fixes in Java projects. We constructed a commit dataset consisting of 103 taint-style vulnerability-fix commits corresponding to 88 CVEs and 1851 non-fix commits across 39 Java projects. EarlyVulnFix was thoroughly evaluated on this collected dataset, yielding promising results. Comparative analysis indicates that EarlyVulnFix significantly outperforms other baselines. Specifically, it achieves a substantially higher F1 score compared to baselines: 62.9% higher than the state-of-the-art deep learning-based approach CoLeFunDa [38] and 950% higher than DAA [10]. Furthermore, when applied to the 700 most recent commits in 7 projects, EarlyVulnFix successfully detected three security fixes before their respective security releases, highlighting its efficacy in detecting unreported silent fixes in the wild.

In summary, this paper makes the following contributions:

- We present EarlyVulnFix, a novel approach designed for the automatic identification of silent taint-style vulnerability fixes. This approach not only considers sanity check fixes but also addresses permission list fixes overlooked in existing research.
- EarlyVulnFix demonstrates significant superiority over state-of-the-art baselines in the identification of silent vulnerability fixes, as evidenced by our evaluation on the collected commits dataset.
- We observe that EarlyVulnFix is capable of identifying vulnerability fixes before the publication of security releases.

2 Background and Motivations

2.1 Background

In this section, we briefly introduce the key concepts used in this paper.

Taint-style vulnerabilities constitute a category of security issues in which untrusted or potentially malicious data flows to security-critical operations (sinks) without proper sanitization. This vulnerability class encompasses various common security issues such as buffer overflows, SQL injections, and deserializations. Among the 2023 CWE Top 25 Most Dangerous Software Weaknesses [2], 12 types fall under the category of taint-style vulnerabilities, with the top 5 all belonging to this specific vulnerability class. This underscores the prominence and critical nature of taint-style vulnerabilities in the realm of software security.

A **security release** represents a particular version of a software product that incorporates one or more vulnerability fixes. These releases are usually accompanied by release notes providing details about the included security fixes, offering guidance to users on the significance of updating to the latest version.

Coordinated vulnerability disclosure is a model wherein a vulnerability or issue is revealed to the public only after the responsible parties have been given ample time to address and patch the identified vulnerability or issue. This approach is widely adopted in numerous OSS organizations [1, 3, 4]. The deliberate delay in public disclosure is intended to minimize the likelihood of exploitation by malicious actors. However, within the transparent landscape of OSS development, if a vulnerability is silently fixed before the security release, a malicious entity could potentially discover these fixes from the public code repository and exploit the vulnerability in susceptible systems.

```
1. public class OgnlUtil {
2.     public void setValue(final String name, final Map<String, Object> context
3.         final Object root, final Object value){
4.         compileAndExecute(name, context, new OgnlTask<Void>() {
5.             public Void execute(Object tree) throws OgnlException {
6.                 if (isEvalExpression(tree, context)) {
7.                     throw new OgnlException();
8.                 }
9.                 Ognl.setValue(tree, context, root, value);
10.                 return null;
11.             }
12.         });
13.     }
14.     private boolean isEvalExpression(Object tree, Map<String, Object> context) {
15.         if (tree instanceof SimpleNode) {
16.             SimpleNode node = (SimpleNode) tree;
17.             OgnlContext ognlContext = null;
18.             if (context!=null && context instanceof OgnlContext) {
19.                 ognlContext = (OgnlContext) context;
20.             }
21.-            return node.isEvalChain(ognlContext);
22.+            return node.isEvalChain(ognlContext) || node.isSequence(ognlContext);
23.         }
24.         return false;
25.     }
26.}
```

(a) An example of Sanity check fixes.

```
1. public class SubTypeValidator {
2.    static {
3. +      s.add("oracle.jdbc.connector.OracleManagedConnectionFactory");
4. +      s.add("oracle.jdbc.rowset.OracleJDBCRowSet");
5.      DEFAULT_NO_DESER_CLASS_NAMES = Collections.unmodifiableSet(s);
6.    }
7.    /**
8.    * Set of class names of types that are never to be deserialized.
9.    */
10.    protected Set<String> _cfgIllegalClassNames = DEFAULT_NO_DESER_CLASS_NAMES;
11.    public void validateSubType(DeserializationContext ctxt, JavaType type) {
12.        do {
13.        // other codes
14.            if (_cfgIllegalClassNames.contains(full)) {
15.                break;
16.            }
17.        } while (false);
18.        throw JsonMappingException();
19.    }
20.}
21.
   public final class TypeFactory implements java.io.Serializable {
22.    protected Class<?> classForName(String name) {
23.        return Class.forName(name);
24.    }
25.}
```

(b) An example of permission list fixes.

Figure 1: Different types of fixes for taint-style vulnerabilities

2.2 Motivating Examples

Figure 1a illustrates a commit addressing a command injection vulnerability within the Struts project, identified by CVE identifier CVE-2016-3081: "when Dynamic Method Invocation is enabled, allow remote attackers to execute arbitrary code via method: prefix, related to chained expressions". The initial code lacked proper input validation, allowing untrusted input data, denoted as *tree*, to reach a critical function call (sink) in line 9, potentially resulting in arbitrary code execution. To mitigate this issue, the developer introduced a validation condition for the variable *node* (a typecast of *tree*) in line 22. If the application encounters malicious input, the *isEvalExpression(tree, context)* function returns true in line 6, triggering an exception and preventing the execution of the dangerous function call.

The fix presented in Figure 1a represents a common way for addressing vulnerabilities, known as sanity checks in a prior study [32]. Taint-style vulnerabilities arise when user-provided data flows to sinks without proper validation, and sanity checks directly examine, sanitize, or escape user-provided data before it reaches sinks. It can be inferred that there are data flow connections between sanity check fixes and sinks. The red arrows in Figure 1a depict the data flow within the program. For instance, data in *tree* in line 6 could flow to the method parameter *tree* in line 14 through a method call, resulting in edges between these two expressions. Through an intermediate statement in line 6, the data in the sanity check establishes a data flow connection with the sink in line 9.

> ***Observation 1.*** *The newly introduced code in sanity checks has a connection with sinks through the data flow of programs.*

Figure 1b shows a commit for the CVE record CVE-2018-12022, reporting that "When Default Typing is enabled, the service has the Jodd-db jar in the classpath, and an attacker can provide an LDAP service to access, it is possible to make the service execute a malicious payload". This commit enhances security by adding two elements to a permission list. If user input contains prohibited class types, a conditional statement in line 14 (s14) detects it and triggers an exception in line 18, preventing potentially dangerous operations in line 23 (s23), which represents the sink of the deserialization vulnerability.

This fix represents another common approach for mitigating taint-style vulnerabilities—adding elements to permission lists, referred to as a "permission list fix" in this paper. Programs utilizing permission lists typically incorporate checks related to these lists, such as the Java method `validateSubType` in Figure 1b. These checks assess the validity of input data or configuration to ensure compliance with rules specified in permission lists. Regardless of the

specific checks applied, they commonly involve conditional statements (e.g., s14) influencing the execution of sinks. In Figure 1b, red edges signify data dependencies, while blue edges denote control dependencies. For instance, the conditional statement s14 utilizes a variable defined in statement line 10 (s10) and controls the execution of s23. Therefore, s14 is data-dependent on s10, and s23 is control-dependent on s14. Notably, the variable *s*, representing the permission list, is linked to the sink s23 through both data and control dependency edges.

> ***Observation 2.*** *Programs that employ permission lists typically incorporate checks associated with these lists. The permission list variables establish connections with sinks through data dependency and control dependency.*

Some approaches for identifying vulnerability fixes have been proposed recently. DAA [10] fails to identify both fixes. In the case of the fix depicted in Figure 1b, the underlying SAST tools fail to detect the corresponding vulnerability. In the fix depicted in Figure 1a, despite the SAST tools being able to generate a security alert for the vulnerability, the alert persists after the fix is introduced.

Existing deep learning-based approaches struggle to learn comprehensive fix patterns due to insufficient context awareness. It can be observed that sinks are located in different functions from fixes, and existing approaches only can capture the surrounding context of fixes within the same functions. Furthermore, providing only probabilities of a commit being a vulnerability fix offers limited assistance to human analysts. In contrast, if the connections between newly introduced code and sinks are detected to identify vulnerability fixes, the process is not constrained by the aforementioned limitations and can provide explanatory insights.

> ***Observation 3.*** *Existing methods for silent vulnerability fix identification exhibit various limitations. Leveraging connections between newly introduced code and sinks can help address these limitations.*

2.3 Key Ideas

From the observations, we have developed EarlyVulnFix with the following key ideas to identify silent vulnerability fixes:

1) **Analyzing data Flow for Identifying Sanity Check Fixes:** Instead of relying on deep neural networks to learn fix patterns, we analyze data flow within the program following Observations 1 and 3. This involves identifying juncture statements, modeling the data flow graph of the program and detecting data flow connections between newly introduced code and program sinks using graph-based techniques. If there are sinks with data flow connections to newly introduced code, our approach identifies the commit as a sanity check fix.

2) **Dependency analysis for Identifying Permission List Fixes:** When newly introduced code in commits includes the addition or creation of collections, our approach utilizes dependency analysis to ascertain whether these collections serve as permission lists. This process entails identifying sinks that could potentially be influenced by these collections. We gather conditional statements that exhibit data dependence on these collections and subsequently identify sinks that display control dependence on these conditional statements. If such sinks exist, EarlyVulnFix identifies the commit as a permission list fix.

3 Proposed Approach

Figure 2 illustrates the overall framework of our approach, EarlyVulnFix, for identifying vulnerability fixes. EarlyVulnFix operates by taking a commit from a software repository and the associated JAR files after that committing as input. It then proceeds to detect connections between the newly introduced code within the commit and sinks within the software. If such connections are found, EarlyVulnFix generates corresponding outputs, specifically the pairs of expressions in the newly introduced code and their corresponding sinks.

To accomplish this, EarlyVulnFix begins by scanning sinks in software using predefined rules maintained by security experts. Subsequently, it determines whether the added code contains sanity checks by analyzing data flow within the software. Additionally, if the added code involves collection addition or creating operations, EarlyVulnFix performs dependency analysis to establish whether this collection serves as a permission list.

3.1 Sink Scanning

As mentioned above, our approach links newly introduced code to sinks, and the initial step is to identify all the sinks within the projects. CodeQL [15] provides a set of rules curated by security experts that define the characteristics of these sinks. Figure 3 presents illustrative examples of these rules. Each rule comprises a list of elements, and each element specifies sink features such as its package, class, method name, argument types, or associated vulnerability types ("command injection" in the example).

EarlyVulnFix initially parses all sink rules and then conducts a systematic examination of all invoke expressions within the projects. With each invocation, our approach assesses whether it matches with a specific rule by considering the package, class, method name, and argument type. It's worth highlighting that the third element in each rule serves as a crucial indicator, as it determines whether subclasses of the rule's class should also be taken into account for a potential match.

3.2 Sanity Checks Identification

Juncture Identification. Building upon Observations 1 and 3 in Section 2.2, we delve into the analysis of data flow connections between newly introduced code and sinks. Apart from the newly introduced statements themselves and those containing sinks, there exists another category of statements crucial for identifying data flow connections. We refer to these statements as "juncture statements." These juncture statements serve as the starting point for our analysis.

The reason behind selecting junctures as our analysis entry, rather than sanity check fixes, lies in the fact that there may not be a direct data flow connection between sanity check fixes and sinks. For instance, in Figure 1a, the fix in Line 22 does not directly connect with the sinks in Line 9. Therefore, we opt to identify the juncture statement in Line 6 as our analysis entry. Specifically, a statement *s* is a juncture statement if and only if the following two conditions

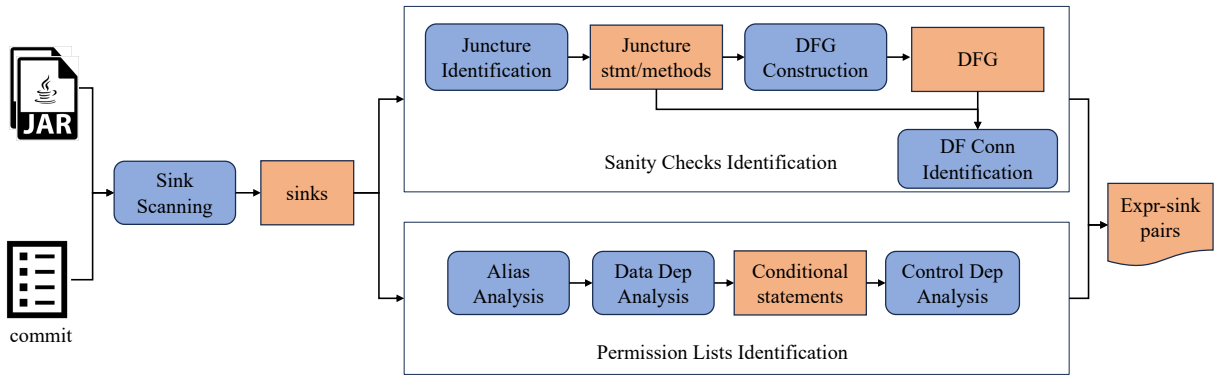

Figure 2: Framework of EarlyVulnFix.

```
- ["java.lang", "ProcessBuilder", False, "ProcessBuilder", "(String[])", "", "Argument[0]", "command-injection", "ai-manual"]
- ["java.lang", "Runtime", True, "exec", "(String,String[])", "", "Argument[0]", "command-injection", "ai-manual"]
- ["java.lang", "Runtime", True, "exec", "(String[],String[])", "", "Argument[0]", "command-injection", "ai-manual"]
```

Figure 3: Examples of sink rules in CodeQL.

are satisfied: 1) s is either a newly introduced statement in commits or an invoke statement that can reach functions containing newly introduced statements in the call graph; 2) there exists a sink that has the potential to be executed after s.

Algorithm 1 shows the process of identifying juncture statements. The input of the algorithm is the statements set *Sinks* containing sinks, statements set *Additions* containing newly introduced statements and call graph *CG* generated by Class Hierarchy Analysis (CHA) [9]. And the output is *Juncs*, denoting the set of juncture statements. The algorithm first initializes the output set

Algorithm 1 Identifying Juncture Statements

Input: *Sinks* is the set of statements containing sinks, *Additions* is the set of newly introduced statements, *CG* is call graph of program.
Output: *Juncs* is a set of juncture statements.

```
1:  Juncs ← ∅
2:  foreach s, addition ∈ Sinks × Additions do
3:      JuncMs ← findJunctureMethods(s, addition, CG)
4:      foreach juncM ∈ JuncMs do
5:          invokeSet1 ← InvokeStatementsOf(juncM, s, CG)
6:          invokeSet2 ← InvokeStatementsOf(juncM, addition, CG)
7:          foreach invokeS, invokeA ∈ invokeSet1 × invokeSet2 do
8:              if invokeA precedes invokeS in control flow graph of
                juncM then
9:                  Juncs ← Juncs ∪ invokeA
10:             end if
11:         end for
12:     end for
13: end for
```

(Line 1). For each sink and newly introduced statement, the algorithm first finds juncture methods that could reach both a method containing the sink and a method containing the newly introduced statement in the call graph through reverse traversal of CG (Lines 2-3). Next, for each juncture method, the algorithm gets invoke statements that can respectively invoke the method containing the sink (*invokeSet*1) and the method containing the newly introduced statement (*invokeSet*2) directly or indirectly (Lines 5-6). If there exists a statement *invokeA* in *invokeSet*2 before another statement *invokeS* in *invokeSet*1 in the control flow graph of the juncture method *juncM*, the algorithm adds this statement to the final set (Lines 7-11). In this context, "before" in the algorithm signifies that *invokeA* can potentially be executed prior to *invokeS*, based on whether *invokeA* can reach *invokeS* in the control flow graph. This step is taken because the execution of sanity checks should occur before the execution of sinks.

Data Flow Graph Construction. A data flow graph models the dynamic flow of data within a program. In this representation, nodes within the graph symbolize program elements responsible for carrying values, such as local variables or static fields. The edges within this graph represent the way data flows between program elements. For example, an assignment statement can transfer data from the right-hand side to the left, resulting in an edge connecting the right-hand variable to the left variable within the data flow graph.

While it is possible to construct precise data flow graphs using pointer analysis techniques [20], or even unify taint analysis with pointer analysis [17, 29], we opted not to employ pointer analysis methods. These techniques often hinge on the presence of a "main" method within the program, which serves as the analysis entry point. In situations where the main method is absent, these tools necessitate manual specification of entry points. The selection of entry points must be careful, as arbitrary entry points may inadvertently skip allocation statements, resulting in empty point-to

Table 1: The seven types of data flow graph edges

	Assignment	Store	Load	Return	This	Parameter	Taint transfer
Statement	$x = y$	$x.f = y$	$y = x.f$	$r = x.k(a1, ..., an)$			
Edge	$x \leftarrow y$	$\_.f \leftarrow y$	$y \leftarrow \_.f$	$r \leftarrow m_{ret}$	$m_{this} \leftarrow x$	$m_{p1} \leftarrow a1$ ... $m_{pn} \leftarrow an$	$r \leftarrow x$ $x \leftarrow a_i$ $r \leftarrow a_i$

sets. In some instances, this process may even require the creation of mock objects to represent entry point parameters [29]. However, our analysis focuses on open-source software libraries, where manual entry point specification becomes impractical. Additionally, precise pointer analysis can suffer from performance issues when dealing with large-scale projects. Due to these challenges, we have chosen to construct a lightweight data flow graph for future analysis instead.

In our approach, the data flow graph comprises two distinct node types. Firstly, expression nodes represent various entities within the program, including local variables, static fields, and instance fields. Secondly, method summary nodes are employed to model method parameters and return values, facilitating interprocedural analysis. It's noteworthy that we adopt a field-based representation for each instance field, which approximates all objects' instances for each field using a single set, disregarding the instance variable. This field-based representation aligns with common practices in taint analysis [36, 37] and pointer analysis [21], demonstrating its practicality in real-world applications [37]. These nodes are interconnected by seven types of edges that delineate data flow relationships. Table 1 enumerates these seven edge types and their corresponding statements. In this table, m_{ret} denotes the return values of method k, m_{this} represents the 'this' values of k, and m_{pi} signifies the i-th parameter of k. Notably, the "Taint transfer" edge, featured in the last column of Table 1, holds particular significance in security-related tasks, often employed in taint analysis. This edge serves to represent three distinct types of data transfers: (1) from the receiver variable (x in the table) to the return value, (2) from a specified argument to the receiver variable, and (3) from a specified argument to the return value. For instance, in the context of a collection adding operation like $s.add(e)$, data may flow from an argument to the receiver variable. We generate a data flow graph for all code reachable from the junction methods acquired earlier.

Data flow Connections Detection. Using the juncture statements and data flow graph obtained before, the present approach detects data flow connections between newly introduced statements and sinks.

Algorithm 2 illustrates the main procedure for detecting data flow connections. The procedure takes as input the data flow graph *DFG*, the set of juncture statements *Juncs*, the set of statements containing sinks *Sinks* and the set of newly introduced statements *Additions*. The desired output is a set of results (*Results*) representing data flow connections. The algorithm iterates through combinations of juncture statements, statements containing sinks, and newly introduced statements (Line 2). For each combination, it further delves into three program elements in three statements respectively, and investigates data flow relationships between these

Algorithm 2 Detecting data flow connections

Input: *DFG* is the data flow graph, *Juncs* is the set of juncture statements, *Sinks* is the set of statements containing sinks, *Additions* is the set of newly introduced statements.
Output: *Results* is the set of data flow connection results.

```
1:  Results ← ∅
2:  foreach junc, sink, addition ∈ Juncs × Sinks × Additions do
3:    foreach e, e′, e′′ ∈ junc × sink × addition do
4:      if hasDataflow(e, e′, DFG) then
5:        if hasDataflow(e, e′′, DFG) or
                hasDataflow(e′′, e, DFG) then
6:          ResultSet ← ResultSet ∪ (e, addition, sink)
7:        end if
8:      end if
9:    end for
10: end for
```

elements (Line 3). For each combination of program elements, the procedure assesses whether a given program element (e) within the juncture statement can establish a path to another program element (e') within the sink statement, utilizing the traversal of the data flow graph (*DFG*). If this connection is established, the procedure subsequently evaluates whether e can also reach the program element (e'') within the newly introduced statement or vice versa. We apply dual-directional analysis here because there may be two distinct pathways through which newly introduced code can influence data. These pathways may involve user-provided data transferring from a juncture statement to a newly introduced statement via function argument passing, or alternatively, data may flow in the reverse direction through the return value of a function.

3.3 Permission List Identification

When our approach detects statements that add elements or create collections in newly introduced code, it determines whether these statements are used to add elements to permission lists or to create permission lists for addressing vulnerabilities. We have observed that programs employing permission lists typically include conditional statements that use these lists to check if current conditions or configurations meet the requirements for executing subsequent code. If these conditions are not met, the statements containing sinks are not executed. Therefore, if the target collection is a permission list, a control dependency exists between the conditional statements and the potential sinks. In the following sections, we will outline the specific procedure for identifying and analyzing this dependency relationship.

Alias analysis. Variables representing permission lists may have multiple aliases across different classes or functions. For instance, consider the variable *s* denoting a permission list, as depicted in Figure 1b, which has two aliases: *DEFAULT_NO_DESER_CLASS_NAMES* and *_cfgIllegalClassNames*. Additionally, in the Intermediate Representation (IR) of the program, each function contains a local variable alias for each instance field. To identify such aliases for collections in newly introduced code, our approach relies on the data flow graph described in Section 3.2. We determine whether two variables are aliases by analyzing their reachability within the data flow graph. If one variable can reach another with the same data type within this graph, we consider them aliases of each other. The reasons for not using precise pointer analysis to obtain aliases are also explained in Section 3.2. We collect all aliases of target collection variables, including the variables themselves, as candidate permission lists for subsequent analysis.

Data dependency analysis. For each candidate permission list identified through alias analysis, the present approach collects conditional statements that are data dependent on it. The process begins by constructing a def-use chain for the method containing the candidate permission list. Subsequently, our approach traverses this def-use chain to identify statements that directly or indirectly utilize the candidate, which we refer to as 'users.' If a user statement is a conditional statement, it is included in the final result. If a user statement is either a return statement or a call statement, our approach recursively gathers conditional statements with the aid of the call graph. To ensure termination and control the complexity of the analysis, we impose a limit on the search depth within the call graph.

Algorithm 3 Analyzing control dependency

Input: *Sinks* is the set of statements containing sinks, *Conditions* is the set of conditional statements obtained by data dependency analysis.
Output: *Controls* is the results of control dependency analysis.

```
1:  Controls ← ∅
2:  foreach condition ∈ Conditions do
3:      method ← condition.getMethod()
4:      CDG ← constructCDG(method)
5:      foreach sink ∈ Sinks do
6:          if method.contains(sink) and
                isDependentOn(CDG, sink, condition) then
7:              Controls ← Controls ∪ (condition, sink)
8:          end if
9:      end for
10:     if isDependentOn(CDG, exception, condition) then
11:         Conduct inter-procedural control dependency analysis
12:     end if
13: end for
```

Control Dependency Analysis. Prior to delving into the detailed process of analyzing control dependencies for the identification of permission list fixes, we establish key definitions related to control dependence based on prior research [13, 18].

Definition 3.1. A node (statement) V is **post-dominated** by a node W in control flow graph G if every path from V to exit node of G contains W.

Definition 3.2. A node (statement) Y is **control dependent** on node X if and only if 1) there exists a path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y; 2) X is not post-dominated by Y.

Intuitively, a statement Y are control dependent on X signifies that X directly decides whether Y executes. Based on our observations, it is common to encounter sink statements that exhibit control dependencies on the conditional statements collected through data dependency analysis in the programs employing permission lists.

The algorithm 3 illustrates the process of analyzing control dependency. It takes two inputs: the set of statements containing sinks *Sinks* , and a set of conditional statements named *Conditions*, obtained through data dependency analysis. The primary objective is to discern and document control dependencies between these conditional statements and the sink statements. The algorithm proceeds by iteratively examining each conditional statement from the *Conditions* set. For each conditional statement, it identifies the method to which it belongs and constructs a Control Dependence Graph (CDG) tailored to that method using the approach proposed by [13] (Lines 3-4). The *CDG* captures the control dependencies within the method. Subsequently, the algorithm scrutinizes each sink statement in the "Sinks" set, seeking control dependency relationships. It checks whether the sink statement resides within the same method as the conditional statement and is control-dependent on the latter, based on the CDG (Line 6). If such a dependency exists, it records the pair (*condition*, *sink*) in the *Controls* set, signifying a control dependency (Line 7).

The algorithm also detects potential inter-procedural control dependencies (Lines 10-12). Inter-procedural control dependence analysis encompasses the identification of three inter-procedural effects: the entry-dependence effect, the multiple-context effect, and the embedded-halt effect, as discussed in [18]. However, the present approach exclusively considers the embedded-halt effect, as it is both prevalent and sufficiently adept for addressing the identification of permission list fixes. For instance, referring to Figure 1b the statement at line 18 serves as an embedded halt. By analyzing its effect, the algorithm can establish a control dependency between the conditional statement at line 14 and the respective sink. The algorithm initiates this process by first determining whether there exists an embedded halt (*exception* in the algorithm) that is control dependent on the current conditional statement (Line 10). If such a relationship is identified, the algorithm proceeds with inter-procedural control dependency analysis. To achieve this, our approach conducts a reverse depth-first traversal of the call graph to identify call sites that may potentially not return due to the presence of embedded halt statements. Subsequently, the algorithm identifies the sink statements that can potentially execute after these call sites. Finally, it records these sink statements in conjunction with the corresponding conditional statement within the *Controls* set.

3.4 Implementation

We have implemented the approach described above into a tool, also named EarlyVulnFix, designed for the detection of silent vulnerability fixes in Java programs. The current implementation of EarlyVulnFix exploits Soot framework [30] and comprises 4,358 lines of Java code. The execution of our tool is divided into seven Soot packs, each responsible for specific functionalities, including tasks such as data flow graph construction, sink scanning, and control dependency analysis. Notably, certain Soot packs are designed to be reusable for various analyses, such as data flow graph construction.

4 Evaluation

We address the following research questions:

RQ1. How effective is EarlyVulnFix in silent vulnerability fixes identification compared with existing fix detectors?

RQ2. How effective are two modules of EarlyVulnFix in detecting sanity check fixes and permission list fixes respectively?

RQ3. Can EarlyVulnFix early sense the vulnerabilities before security release in the wild?

4.1 Data Collection

In order to evaluate tools for identifying silent fixes in RQ1 and RQ2, it is crucial to possess a dataset that includes labeled commits. Below, we provide a description of the data collection process.

Step 1. Collecting Java vulnerabilities and the corresponding fixes. We acquire Java CVE data along with the corresponding fixes from the dataset compiled in a prior study [39]. This dataset stands out as the most extensive real-world vulnerability benchmark for Java to our knowledge. The Java segment of this dataset comprises 839 CVEs, corresponding to 1436 vulnerability fixes, and encompasses 310 open-source software (OSS) projects.

Step 2. Filtering taint-style vulnerabilities with CWE category. We obtained CWE category information for each CVE from the National Vulnerability Database (NVD) [28]. As our current research focuses on identifying fixes for taint-style vulnerabilities, we specifically chose CVEs falling within four CWE categories associated with taint-style vulnerabilities: CWE-77 (Command Injection), CWE-78 (OS Command Injection), CWE-89 (SQL Injection), and CWE-502 (Deserialization of Untrusted Data). These categories, all part of the top 25 CWEs list[2], are known for their widespread impact and potential for severe consequences. In cases where a CVE is assigned multiple CWE categories, we included it in our selection if at least one of the assigned categories belonged to these four categories. After filtering, 88 CVEs remain in consideration.

Step 3. Collecting non-fix commits It is challenging to distinguish normal non-fix commits from fix commits accurately. To address this difficulty, we devised an approach to assemble non-fix commits by extending the collection of vulnerability-fix commits outlined in Step 2. Our process starts by associating each vulnerability-fix commit with the specific security release that incorporates it. Following this, we manually scrutinize release notes to identify security releases that exclusively address a single vulnerability. Subsequently, we collect all the commits included in these security releases as non-fix commits, excluding the vulnerability-fix commits themselves. This collection approach yields two key advantages. Firstly, it enhances the precision of labeling by leveraging information from release notes. Secondly, it aligns with the usage scenario of tools for silent fix identification. These tools are typically applied to commits after the latest releases to detect vulnerability fixes preceding future security releases. By employing this collection approach, our dataset comprises 103 vulnerability-fix commits and 1851 non-fix commits across 39 Java projects, corresponding to 88 CVEs.

Table 2: Performance of EarlyVulnFix and deep-learning based approaches in the vulnerability fix identification task

Approach	TP	FP	TN	FN	Rec.	Prec.	F1
VulFixMiner	4	3	903	17	19.0%	57.1%	0.29
CoLeFunDa	10	26	880	11	47.6%	27.8%	0.35
EarlyVulnFix	16	19	856	5	76.2%	45.7%	0.57

Table 3: Performance of EarlyVulnFix and DAA in the vulnerability fix identification task

Approach	TP	FP	TN	FN	Rec.	Prec.	F1
DAA	2	0	505	46	4.2%	100.0%	0.08
EarlyVulnFix	46	16	475	2	95.8%	74.2%	0.84

4.2 RQ1: Comparison With State-of-the-Art Silent Fix Identification Approaches

Baselines. We compare EarlyVulnFix with three baseline approaches: **VulFixMiner** [39], **CoLeFunDa** [38] and **DAA** [10]. **VulFixMiner** and **CoLeFunDa** are deep learning-based approaches that learn commit-level code change representations and function-level patch representations, respectively, to detect silent vulnerability fixes. **DAA** is an algorithm designed to identify resolved vulnerabilities in software projects by harnessing the outputs of off-the-shelf SAST tools. For each commit, DAA performs SAST analysis on two consecutive versions of a software project and detects resolved vulnerabilities by identifying the removal of SAST alerts present in the previous software versions.

Methodology. To facilitate a comparison with the deep learning-based approaches, we reached out to the author of VulFixMiner and CoLeFunDa to obtain their model. Our dataset comprises 103 vulnerability-fix commits and 1851 non-fix commits. However, 82 vulnerability-fix commits and their corresponding non-fix commits were already part of the training set for provided models. To ensure a fair comparison, we exclusively compared EarlyVulnFix with VulFixMiner and CoLeFunDa in the remaining 21 vulnerability-fix commits and their associated 875 non-fix commits.

While DAA does not rely on deep learning and does not require a training set, it necessitates that the project can be successfully compiled to obtain the results of static analyzers. In certain older projects, dependencies have been removed from their respective repositories, making compilation impossible and preventing DAA from running on these projects. As a result, we could only compare our tool to DAA on 48 (out of 103) vulnerability-fix commits and 491 associated non-fix commits from 10 projects that compiled

successfully. We use CodeQl [15] as the underlying off-the-shell static analyzer for DAA, which is consistent with the original DAA work.

To evaluate the effectiveness of various approaches, we employ the following metrics: Precision, calculated as $Precision = \frac{TP}{TP+FP}$; Recall, calculated as $Recall = \frac{TP}{TP+FN}$; and F1-score, calculated as $F1-score = \frac{2\times Precision\times Recall}{Precision+Recall}$. Here, TP represents True Positives, FP denotes False Positives, FN signifies False Negatives, and TN stands for True Negatives.

Results. The performance results of our tool and deep learning-based approaches are presented in Table 2. VulFixMiner achieved a recall rate of 19.0%, CoLeFunDa demonstrated a recall of 47.6%, while EarlyVulnFix excelled with an impressive recall of 76.2%. Furthermore, EarlyVulnFix exhibited a competitive F1 score of 0.57, outperforming VulFixMiner and CoLeFunDa by 96.9% and 62.9%, respectively. These results highlight the challenges faced by deep learning-based approaches, as discussed in Section 1, leading to relatively low recall and F1 scores. In contrast, our tool examines data flow and dependency connections between newly introduced code and sinks, overcoming the limitations of deep learning techniques. For instance, in addressing CVE-2017-12612, a high-risk deserialization vulnerability in the Apache Spark project, the newly introduced code in the fix validates the descriptor of a class before deserialization. Our tool accurately detects the data flow from the newly introduced code to the deserialization sink, providing interpretable results. However, VulFixMiner and CoLeFunDa yield a low probability score for this commit due to insufficient context awareness.

As illustrated in Table 3, EarlyVulnFix successfully identifies 46 out of 48 vulnerability-fix commits, while DAA can only identify 2 out of the 48. The F1 score for EarlyVulnFix is 0.84, while DAA achieves a lower score of 0.08. EarlyVulnFix significantly outperforms DAA in recall while maintaining high precision (74.2%). The poor performance of DAA is attributed to the inherent limitations of the underlying static analyzer, which fails to detect most vulnerabilities before the application of security fixes. Consequently, DAA is unable to identify the removal of alerts in the fixed version. Our findings align with a recent empirical study [24] evaluating Java static application security testing tools, highlighting the challenges faced by existing tools in detecting real-world vulnerabilities due to the lack of efficient rules, including sources, sinks, and sanitizing rules. In contrast, EarlyVulnFix relies only on the sink rules of the existing static analyzer, which, based on our observations, is relatively comprehensive. It's noteworthy that there are still 2 fixes that cannot be detected by EarlyVulnFix, and we will defer the discussion about the limitations of EarlyVulnFix to Section 4.3.

4.3 RQ2: Detection on Different Types of Fixes

As described in Section 3, our approach involves the identification of two distinct types of fixes within our approach: sanity check fixes and permission list fixes. The former focuses on sanitizing or validating user-provided data before it engages in security-critical operations, while the latter involves adding or creating a permission list. These two fix types are addressed through separate modules. This section presents an evaluation of the effectiveness of our tool's two modules in detecting the two types of fixes respectively.

Table 4: Performance of two modules of EarlyVulnFix in the vulnerability fix identification task

Module	TP	FP	TN	FN	Rec.	Prec.	F1
Sanity	27	21	1221	11	71.1%	56.3%	0.64
Permission	40	0	375	11	78.4%	100.0%	0.88
Total	66	22	1230	16	80.5%	75.0%	0.78

Methodology. Our tool relies on JAR files containing newly introduced code from commits, as detailed in Section 3. In instances where projects publish their compiled JAR files to Maven repositories, we directly download these files. For other projects, we compile the source code to generate the required JAR files, excluding projects that either do not release JAR files or fail to compile successfully. Our evaluation dataset in RQ2 comprises 82 vulnerability-fix commits and 1297 non-fix commits.

To categorize the vulnerability-fix commits, we manually inspect their content and classify them as either sanity check fixes or permission list fixes. Of the total, 38 commits are identified as sanity check fixes, while 51 are recognized as permission list fixes. Eight commits fall into both categories, as they not only create permission lists but also utilize these lists to sanitize user-provided data. Additionally, one commit does not align with either fix type, as it solely involves the removal of code without introducing new code. The application of our two modules is thus tailored to the specific fix types, such as applying the Sanity Check Identifications module exclusively to commits labeled as sanity check fixes.

Result. Table 4 shows the effectiveness of two modules of EarlyVulnFix in detecting sanity check fixes and permission list fixes respectively and the performance of EarlyVulnFix as a whole on the overall dataset.

The second row of the table indicates that among 38 sanity check fixes, the corresponding module successfully identifies 27 of them, while erroneously flagging 21 out of 1242 non-fix commits as security fixes. The module performs admirably in detecting sanity check fixes, exhibiting high recall, precision and F1 score. However, it does face limitations in certain scenarios. In 11 false-negative cases, 4 cases fail because sinks are not included in the rules of CodeQl, and 7 cases fail because specific data flow steps cannot be modeled by the data flow graph of the program, as illustrated in the situation where a command flows to a sink through a file in the fix for CVE-2020-35476. Additionally, the module of EarlyVulnFix may generate false positives, such as in some non-fix commits where developers alter program functionality and manipulate variables that could flow to sinks. For instance, in commit 0de92b1 within the plexus-utils project, developers introduce a feature to enable asynchronous execution of commands. The data manipulated by the newly introduced code erroneously flows to the sink of command injection, leading to the incorrect identification of the commit as a security fix by our tool.

The third row of the table indicates that among 51 permission list fixes, the corresponding module for identifying permission list fixes successfully identifies 40 of them. The module achieves a precision of 100%, exhibiting no false positives in the 375 non-fix commits. The high precision is attributed to the strict conditions that a commit must meet to be classified as a security fix by this

module, as explained in Section 3.3. However, this module may encounter challenges in certain situations. In 11 false-negative cases, 4 cases fail because they append permission lists using different methods, such as adding files to a text file containing permission list terms without modifying the code. In addition, 2 cases fail due to problems with the current implementation, and 5 cases fail because conditional statements impact sinks through implicit data flow. Tracking implicit flows could result in significant performance overhead and a decline in precision [11, 31]. Therefore, we choose to disregard implicit data flow, aligning with existing taint tools [5, 11].

The last row of Table 4 presents the individual performance of EarlyVulnFix on the dataset encompassing all analyzable commits. Out of the 82 vulnerability-fix commits, EarlyVulnFix accurately identifies 66, while erroneously flagging 22 out of the 1297 non-fix commits as security fixes. The overall performance metrics on the dataset show that EarlyVulnFix achieves an 80.5% recall rate, a 75.9% precision rate, and a F1 score of 0.78. Additionally, during our investigation in the *Pippo* project, we noticed two commits labeled as non-fix, which EarlyVulnFix identified as vulnerability-fix commits. Subsequent analysis revealed that these were "polish" vulnerability fixes occurring after the initial fix, and they were not documented in the CVE database. This underscores EarlyVulnFix's ability to unveil previously undocumented vulnerability fixes.

4.4 RQ3: Early Vulnerability Sensing in the Wild

Methodology. In this research question, we assess the effectiveness of EarlyVulnFix in proactively identifying vulnerabilities in real-world scenarios, rather than relying on historical vulnerability datasets. To do so, we selected projects that exhibited a notable presence of taint-style vulnerabilities within the past two years. This selection was based on information from the open-source vulnerability database [16, 27].

Ultimately, we identified 7 projects for our evaluation: *Onedev, Reload4j, OpenRefine, Apache InLong, Apache Zeppelin, Yamcs, Apache Shardingsphere*. We systematically scanned the 100 most recent commits (prior to November 1, 2023) for each of these projects using our tool. Any commits flagged as vulnerability fixes by our tool underwent manual confirmation by three security experts from a prominent IT enterprise, each possessing at least five years of experience in software security.

Results. In the analysis of the 700 most recent commits across 7 projects, EarlyVulnFix identifies 31 commits as vulnerability fixes. Of these 31, security experts confirm 7 as addressing security vulnerabilities. Notably, 3 of these security fixes are detected prior to the corresponding security release, while the remaining 4 are already incorporated into a released security update. It is noteworthy that one fix among the remaining 4 pertains to a vulnerability not reported to CVE. These results demonstrate the tool's effectiveness in early detection of vulnerabilities, offering users of open-source software more time and improved preparedness against potential security threats prior to the official publication of security releases or advisories.

Case study. EarlyVulnFix successfully identified fixes for a SQL injection vulnerability in *Apache Zeppelin*. At the time of writing, the corresponding security release had not been published, and the vulnerability had not been reported to CVE. The fix involved parameterizing the SQL query to prevent SQL injection vulnerabilities. Our tool effectively traced the data flow from the SQL query to the SQL injection sink.

EarlyVulnFix successfully identified fixes for CVE-2023-45278, committed on October 9, 2023. The detection occurred on October 27, 2023, predating the publication of security release 5.8.8 on November 3, 2023. This vulnerability involves a directory traversal issue, allowing attackers to delete arbitrary files by exploiting a crafted HTTP DELETE request. To address this, the fix introduces code that validates user-provided objects and raises an exception when the object contains malicious content. This fix represents a typical sanity check solution, and EarlyVulnFix identified the data flow from the newly added code to the sink.

5 Discussion

5.1 Other Possible Application Scenarios

Based on its capability to detect silent vulnerability fixes, EarlyVulnFix can be extended to various security tasks. For instance, it can enhance published vulnerability information by identifying the corresponding fixes, which is crucial for automatically generating security patches. Additionally, EarlyVulnFix can be used for vulnerability dependency alerts (e.g. GitHub Dependabot [14]), enabling users to receive alerts at an earlier stage.

5.2 Ethical Consideration

Cybersecurity is always an arms race and often a two-edged sword. Different from solutions used by malicious parties, our solution will be open and is created with the intent to protect. Our solution is the same as many other solutions (e.g., fuzzing [40] and automatic exploit generation [6]), which can be used by both malicious parties and OSS users. However, such research is still valuable as it empowers OSS users to defend against potential attacks.

5.3 Threats to Validity

Internal validity: Threats to internal validity relate to experimenter bias and errors. To mitigate bias in the collection of vulnerability fix commits, we rely on an existing real-world vulnerability benchmark as the foundation for dataset construction. Labeling non-fix commits poses another threat to validity. Adopting an approach where all commits are labeled as non-fix except for the vulnerability fix commits could lead to mislabeling security fix commits due to the absence of fix data in the vulnerability database. To address potential errors in the dataset, we label non-fix commits only within the same releases as security fixes, utilizing release notes for assistance. Additionally, to minimize bias in manually examining cases in RQ3, we engage security professionals with a minimum of 5 years of experience in software security and provide ample time for manual checks.

External validity: Threats to external validity pertain to the generalizability of EarlyVulnFix. To mitigate these threats, we conduct experiments to evaluate the effectiveness of EarlyVulnFix by applying it to the 700 most recent commits of 7 projects outside the dataset we constructed.

6 Related Work

6.1 Early Vulnerability Sensing

Early vulnerability sensing involves identifying their presence before they are publicly disclosed. In open-source software, it's common for vulnerabilities to be silently fixed without explicit commit messages revealing the issue. To address this, early sensing tools are essential for recognizing fixes without relying on textual information, a process known as silent fixes identification. Early vulnerability sensing is valuable as it can substantially decrease the resources needed to locate and address vulnerabilities, preventing their exploitation. Consequently, numerous efforts have been made to develop methods for identifying silent fixes in order to enhance software security.

VulFixMiner [39] is a Transformer-based approach that automatically detects silent vulnerability fixes by extracting semantic meaning from code changes at the commit level. VulFixMiner demonstrates increased efficiency in identifying silent vulnerability fixes and has the capability to recognize unreported fixes.

CoLeFunDa [38] is a deep learning-based framework designed to identify silent fixes and offer explanations for the identified silent fixes. CoLeFunDa utilizes data augmentation techniques to predict not just whether a commit is a security fix but also predicts the CWE category and exploitability rating, providing results with explanations. In comparison to CoLeFunDa, our approach goes beyond offering only the CWE category; it also reveals the data flow or dependency connection between newly added code and sinks, aiding in future manual inspections.

SPAIN [32] is a scalable framework for binary-level patch analysis, capable of automatically identifying security patches and summarizing patch patterns based on the original and patched versions of a binary program. SPAIN initially identifies changed traces and subsequently conducts a semantic analysis of these traces to pinpoint security patches. In contrast, our approach specializes in the identification of security fixes within the open-source ecosystem.

DAA [10], Differential Alert Analysis, is a recently proposed approach for uncovering vulnerability fixes in software projects. Let P represent a software project, and P' denote the subsequent version of P. DAA utilizes an off-the-shelf static analyzer to generate security-related alerts for both P and P'. Subsequently, it extracts alerts present in P but not in P'. Finally, DAA introduces a novel process to automate the announcement level for the identified fixes.

6.2 Study on Security Patches

Li and Paxson [23] conducted a comprehensive empirical study covering 4,000 security patches. Their findings reveal that security fixes are frequently publicly visible for weeks before their official announcement. They emphasize the potential risks associated with delays in disclosure.

Imtiaz et al. [19] examined security releases and fixes in open-source packages. Their findings reveal that one-fourth of the releases occurred at least 20 days after the fix was implemented. Additionally, only 61.5% of security releases included a release note documenting the security fix. This underscores the importance of real-time, automated identification of vulnerability fixes.

Chinthanet et al. [8] investigated the delays in the release, adoption, and propagation of npm vulnerability fixes. Their findings indicate that a security release is seldom released independently, with as much as 85.72% of bundled commits being unrelated to a fix. This factor contributes to delays between the security fix and its subsequent release. The study also explores the speed at which security fixes propagate within the npm ecosystem.

7 Conclusion and future work

In this paper, we propose EarlyVulnFix, for silent taint-style vulnerability identification using program analysis technique. The core principle guiding our method is the recognition that newly introduced code in fixes is interconnected with vulnerability sinks. EarlyVulnFix begins by scanning the project's code to identify sinks, utilizing established rules from existing SAST tools. Subsequently, our tool utilizes a lightweight approach to analyze the data flow to detect sanity check fixes and conducts inter-procedural dependency analysis to identify permission list fixes. Ultimately, we present the identified connections to human analysts, facilitating their manual analysis. Evaluation results demonstrate that EarlyVulnFix significantly outperforms state-of-the-art baselines. Moreover, when applied to the 700 latest commits, EarlyVulnFix proves its capability to identify vulnerability fixes before the publication of security releases.

For future work, we plan to explore various strategies to enhance our approach, such as supporting permission lists stored in text files, addressing the limitations discussed in Section 4.3. Additionally, we aim to extend the generalizability of our approach to encompass more programming languages like PHP and C/C++, which also contend with taint-style vulnerabilities.

8 Data Availability

Our code and data are available at:
https://github.com/wzzll123/EarlyVulnSense.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research was supported by the National Natural Science Foundation of China under Grant Nos. 62372227 and 62232014.

References

[1] 2022. *Coordinated vulnerability disclosure policies in the eu.* https://www.enisa.europa.eu/news/enisa-news/coordinated-vulnerability-disclosure-policies-in-the-eu [online].

[2] 2023. *2023 CWE Top 25 Most Dangerous Software Weaknesses.* https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html [online].

[3] 2023. *About coordinated disclosure of security vulnerabilities.* https://docs.github.com/en/code-security/security-advisories/guidance-on-reporting-and-writing-information-about-vulnerabilities/about-coordinated-disclosure-of-security-vulnerabilities [online].

[4] 2023. *Microsoft's Approach to Coordinated Vulnerability Disclosure.* https://www.microsoft.com/en-us/msrc/cvd [online].

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[6] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (feb 2014), 74–84. https://doi.org/10.1145/2560217.2560219

[7] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and

Implications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 143–157. https://doi.org/10.1109/SP.2008.17

[8] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empir. Softw. Eng.* 26, 3 (2021), 47. https://doi.org/10.1007/S10664-021-09951-X

[9] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 952)*, Walter G. Olthoff (Ed.). Springer, 77–101. https://doi.org/10.1007/3-540-49538-X_5

[10] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. 2023. Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*. IEEE, 489–505. https://doi.org/10.1109/EuroSP57164.2023.00036

[11] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2 (2014), 5:1–5:29. https://doi.org/10.1145/2619091

[12] Douglas Everson, Long Cheng, and Zhenkai Zhang. 2022. Log4shell: Redefining the web attack surface. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022*.

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[14] Github. 2019. *Automated security updates*. https://github.blog/changelog/2019-11-14-automated-updates/ [online].

[15] GitHub. 2023. *CodeQL*. https://codeql.github.com/ [online].

[16] Google. 2023. *OSV - Open Source Vulnerabilities*. https://nvd.nist.gov/vuln/data-feeds [online].

[17] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 102:1–102:28. https://doi.org/10.1145/3133926

[18] Mary Jean Harrold, Gregg Rothermel, and Saurabh Sinha. 1998. Computation of Interprocedural Control Dependence. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 1998, Clearwater Beach, Florida, USA, March 2-5, 1998*, Mary Lou Soffa, Michal Young, and Will Tracz (Eds.). ACM, 11–20. https://doi.org/10.1145/271771.271780

[19] Nasif Imtiaz, Aniqa Khanom, and Laurie A. Williams. 2023. Open or Sneaky? Fast or Slow? Light or Heavy?: Investigating Security Releases of Open Source Packages. *IEEE Trans. Software Eng.* 49, 4 (2023), 1540–1560. https://doi.org/10.1109/TSE.2022.3181010

[20] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 291–302. https://doi.org/10.1145/2737924.2737957

[21] Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[22] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2201–2215. https://doi.org/10.1145/3133956.3134072

[23] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2201–2215. https://doi.org/10.1145/3133956.3134072

[24] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. (2023), 921–933. https://doi.org/10.1145/3611643.3616262

[25] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 544–555. https://doi.org/10.1145/3533767.3534380

[26] Benjamin Livshits and Stephen Chong. 2013. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. *SIGPLAN Not.* 48, 1 (jan 2013), 385–398. https://doi.org/10.1145/2480359.2429115

[27] NIST. 2023. *National Vulnerability Database*. https://nvd.nist.gov [online].

[28] NIST. 2023. *NVD - data feeds*. https://nvd.nist.gov/vuln/data-feeds [online].

[29] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1093–1105. https://doi.org/10.1145/3597926.3598120

[30] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[31] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. 2023. ConfTainter: Static Taint Analysis For Configuration Options. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1640–1651. https://doi.org/10.1109/ASE56229.2023.00067

[32] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 462–472. https://doi.org/10.1109/ICSE.2017.49

[33] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. 797–812. https://doi.org/10.1109/SP.2015.54

[34] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. 1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 867–879. https://doi.org/10.1145/3597926.3598102

[35] Fang Yu, Ching-Yuan Shueh, Chun-Han Lin, Yu-Fang Chen, Bow-Yaw Wang, and Tevfik Bultan. 2016. Optimal Sanitization Synthesis for Web Application Vulnerability Repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2931037.2931050

[36] Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, and Alex X. Liu. 2022. Field-Based Static Taint Analysis for Industrial Microservices. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 149–150. https://doi.org/10.1109/ICSE-SEIP55303.2022.9794096

[37] Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, Alex X. Liu, and John C. S. Lui. 2023. Scalable Compositional Static Taint Analysis for Sensitive Data Tracing on Industrial Micro-Services. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 110–121. https://doi.org/10.1109/ICSE-SEIP58684.2023.00015

[38] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E. Hassan. 2023. CoLeFunDa: Explainable Silent Vulnerability Fix Identification. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2565–2577. https://doi.org/10.1109/ICSE48619.2023.00214

[39] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E. Hassan. 2021. Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 705–716. https://doi.org/10.1109/ASE51524.2021.9678720

[40] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345

Received 16-DEC-2023; accepted 2024-03-02