PATEN: Identifying Unpatched Third-Party APIs via Fine-grained Patch-enhanced AST-level Signature

Li Lin, Jialin Ye, Chao Wang, and Rongxin Wu

Abstract—Using a third-party library (TPL) API that is still unpatched with respect to known vulnerabilities would introduce severe security threats, and thus it is important to detect unpatched API as early as possible. Existing vulnerability detection methods often fail to identify subtle differences between patched and vulnerable versions of code, leading to high rates of false positives and missed vulnerabilities. Addressing these limitations, we propose a novel approach that employs a fine-grained, patch-enhanced Abstract Syntax Tree (AST) level signature. This approach consists of two key steps: patch-induced AST difference extraction and vulnerability trace refinement. These steps enable the detailed analysis of structural changes due to patches and enhance the accuracy of vulnerability detection by focusing on the critical elements of code changes.

Building on this methodology, we introduce PATEN, a tool designed to accurately detect unpatched TPL APIs. Our evaluation, conducted on a large dataset, demonstrates that PATEN significantly outperforms the state-of-the-art approaches. Specifically, PATEN identified 82 critical vulnerabilities across numerous open-source projects, demonstrating a substantial advancement in the field of unpatched TPL API detection and highlighting its practical implications for improving software security.

Index Terms—Unpatched API, Vulnerability detection and Third-party Libraries

1 Introduction

Vulnerabilities in third-party libraries (TPLs) are widespread [1], [2], [3], [4], [5], [6], [7], making the identification of vulnerable TPL API usage critically important for effective software maintenance [1], [4], [8], [9]. To cater for such a need, some recent studies propose to leverage the call graph analysis to check whether vulnerable TPL APIs [1], [4], [9] are reachable from a given application project. However, these techniques are unable to proceed without knowing whether a TPL API is still unpatched with respect to the known vulnerabilities. These approaches are constrained by the availability of complete and accurate metadata (e.g., library name, affected versions, vulnerability description) [9]. Unfortunately, these metadata, which are used to map each TPL onto a list of known vulnerabilities that affect it, are often incomplete, inconsistent, or missing altogether [1]. For example, the metadata of a vulnerable TPL API is often not available since its corresponding library would often be recompiled, repackaged, or manipulated [1]. This motivates us to study the problem of unpatched API identification without relying on metadata. Our key idea is to leverage a pre-constructed vulnerability database and analyze whether a reachable vulnerable TPL API is still unpatched, based on the specific known vulnerabilities and patches associated with it.

- *Li Lin, Jialin Ye, Chao Wang, and Rongxin Wu are with School of Informatics, Xiamen University, China. E-mail: {linli1210@stu.xmu.edu.cn, yejialin@stu.xmu.edu.cn, wangc@stu.xmu.edu.cn, wurongxin@xmu.edu.cn}. Rongxin Wu is the corresponding author.*

Manuscript received September 13, 2024; revised xxx, 2024.

Existing approachs. Our task is to ascertain whether a given TPL API is vulnerable or has been patched, taking into account specific known vulnerabilities and patches. A general idea for identifying unpatched APIs involves comparing the source code of a target API with the code of known vulnerabilities and their corresponding fixed versions [10], [11], [12], [13], [14], [15], [16], [17]. Existing approaches can be categorized into *clone-based* and *patch-enhanced matching* approaches.

Clone-based approaches [12], [15], [16], [17], [18], [19] consider the identification of unpatched APIs as a code clone detection problem. These methods involve extracting signatures from both vulnerable and patched APIs. The core idea is to compare a target API against these signatures to determine similarity. If the target API is more similar to the signature of a vulnerable API than to that of its patched counterpart, it is identified as vulnerable. However, due to the nature of clone detection and no consideration of how a vulnerability is fixed, clone-based approaches fail to differentiate the potentially small differences between vulnerable API and patched API, causing high false positives [20]. Consequently, these methods are not specifically tailored for the accurate identification of unpatched APIs.

Patch-enhanced matching approaches [10], [11], [14], [20], [21], [22] alleviate the limitations of clone-based approaches by leveraging the characteristics of vulnerability-fixing patches. However, these methods predominantly preserve statement-level characteristics, often using techniques like hashing algorithm to directly encode statements into signatures. This coarse-grained representation often fails to distinguish between critical (i.e., vulnerability-related or patch-related) and non-critical vulnerable elements within statements, limiting the effectiveness in accurately identify-

CVE Number: CVE-2016-0168

Bug Fix Commit: 2d9b168cfbbf5a6d16fa6e8a5b34503e3dc42364

```
154 public Map<String, String> handleCorsPreflightRequest (String pOrigin,
                String pRequestHeaders) {
155             Map<String,String> ret = new HashMap<String, String>();
156   - if (pOrigin != null && backendManager.isCorsAccessAllowed(pOrigin))
157   + if (pOrigin != null && backendManager.isOriginAllowed(pOrigin,false))
      // .....
168 }
```

(a) Patch for fixing CVE-2014-0168 (merged into jolokia-core 1.2.1, 1.2.2, 1.2.3, 1.3.0, 1.3.1, 1.3.2, 1.3.4, 1.3.5, 1.3.6, 1.3.7, 1.4.0, 1.5.0, 1.6.0, 1.6.1, 1.6.2, 1.7.0, 1.7.1 ...)

Target Version: 1.0.3

Bug Fix Commit: 932c7a51d4a251a6c1f03868415ba3c89f5ee205

Is Vulnerable: True

```
134 public Map<String, String> handleCorsPreflightRequest (String pOrigin,
                String pRequestHeaders) {
135     Map<String,String> ret = new HashMap<String, String>();
136     if (backendManager.isCorsAccessAllowed(pOrigin))
137 //  .....
146 }
```

(b) **A vulnerable target TPL API (**in jolokia-core 1.0.3**)**

Fig. 1: Example 1 of vulnerability-fixing patch and a target API that is vulnerable. Highlighting critical vulnerability elements in identifying unpatched APIs.

ing nuanced vulnerabilities.

Motivation example. Figure 1(a) shows a patch for fixing the API for the vulnerability CVE-2014-0168. Figure 1(b) shows a target API that is vulnerable. However, using patch-enhanced matching approaches that rely on coarse-grained representations (i.e., statement-level signatures), the modified line (Line 156) in Figure 1(a) does not match with Line 136 in Figure 1(b), resulting in a false negative. In reality, the critical vulnerable element "`backendManager.isCorsAccessAllowed`" in Line 156 perfectly matches Line 136 in Figure 1(b). This misclassification primarily occurs because the statement-level signature fails to differentiate between the critical vulnerable element "`backendManager.isCorsAccessAllowed`" and the non-critical element "`pOrigin!=null`".

Key insight. To precisely target critical vulnerable elements within patches, we introduce a fine-grained *patch-enhanced AST-level signature* that offers a more detailed description of code structure and characteristics. This approach surpasses the capabilities of statement-level signatures by providing a more accurate localization of vulnerability modifications. The patch-enhanced AST-level signature is realized through the following two main steps:

- *Patch-Induced AST Difference Extraction*: This step involves extracting subtrees related to changes made during the vulnerability fix, enabling the pinpointing of the critical vulnerable element. The AST difference effectively highlights minor modifications by reflecting changes in nodes, such as types and variable names.
- *Vulnerability Trace Refinement*: Based on our observations, the genesis of vulnerabilities is frequently linked to variable propagation. This step refines the extracted subtrees by tracking variable interactions through program slicing, seamlessly integrating context-specific AST segments to form a complete subtree that effectively highlights critical changes. The refined differential subtrees then serve as fine-grained, enhanced signatures, significantly improving the accuracy of vulnerability detection.

Implementation. We developed PATEN, leveraging the patch-enhanced AST-level signature to detect vulnerabilities in software. The process begins by assembling a detailed database of known vulnerabilities, sourced from platforms like *Snyk* [23] and GitHub [24]. Subsequently, we extract signatures for both the vulnerabilities and their patches pertaining to affected APIs. PATEN matches these signatures against the APIs utilized in a given project to pinpoint any that are unpatched or still vulnerable. Finally, to assess the real-world impact, we check if these vulnerable APIs are actually being called within the project.

Evaluation. To evaluate the effectiveness of PATEN, we mined patches for 334 known Java library vulnerabilities from prominent security sources, including the *Snyk* platform [23] and the Github platform [24]. Our ground truth dataset comprises 30,389 versions of TPL APIs, both vulnerable and patched.

We compared PATEN against six leading patch-enhanced matching approaches: VUDDY [11], REDEBUG [10], MVP [20], MOVERY [21], SECURESYNC [14], and VISION [22]. The comparative analysis revealed that PATEN significantly outperformed its competitors, achieving improvements in F1-measure by 20% for VISION, 39% for SECURESYNC, 1,300% for MVP, 3,364% for REDEBUG, 3,586% for MOVERY, and 5,321% for VUDDY, demonstrating its superior capability in accurately detecting vulnerabilities. We also employed it to detect unpatched APIs in real-world projects. Finally, we submitted 82 bug reports, of which 71 are fixed by developers and 11 are already confirmed.

Contributions. We summarize our contributions below.

- We introduce a fine-grained patch-enhanced AST-level signature specifically designed for the identification of unpatched APIs, which provides a more accurate analysis of vulnerabilities.
- We implemented PATEN with the patch-enhanced AST-level signature and conducted a comprehensive evaluation against a large-scale ground truth dataset, revealing that PATEN significantly outperforms existing state-of-the-art techniques in both accuracy and reliability.
- PATEN found 82 bugs that are related to the usage of unpatched TPL APIs in the open source projects.

2 MOTIVATION

In this section, we illustrate the importance of identifying unpatched APIs, analyze the limitations of existing approaches, and present our solution based on a fine-grained methodology.

2.1 Importance of Identifying Unpatched API

The presence of vulnerabilities within TPLs poses significant risks across software ecosystems. These vulnerabilities are increasingly prevalent, underscoring the urgency for effective security measures [25], [26], [27], [28]. Critically, not all vulnerabilities found in TPLs are actively exploited

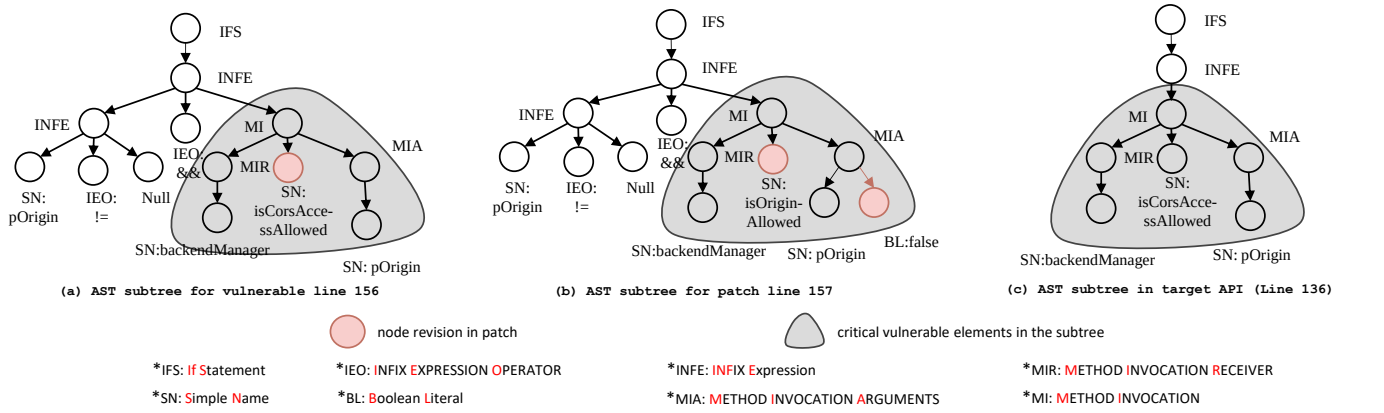


Fig. 2: Solutions of Example 1 through patch-induced AST-Level difference extraction, highlighting critical vulnerable elements within the statement.

CVE Number: CVE-2016-4438

Bug Fix Commit: 76eb8f38a33ad0f1f48464ee1311559c8d52dd6d

```
298  private void handleDynamicMethodInvocation
                    (ActionMapping mapping, String name) {
     // .....
302      String actionMethod = name.substring(exclamation + 1);
     // .....
310  }
311
312      mapping.setName(actionName);
313      if (allowDynamicMethodCalls) {
314  -       mapping.setMethod(actionMethod);
314  +       mapping.setMethod(cleanupActionName(actionMethod));
315      } else {
316          mapping.setMethod(null);
     // .....
319  }
```

(a) Patch for fixing CVE-2016-4438

Target Version: 2.3.24.1

Is Vulnerable: True

```
203  private void handleDynamicMethodInvocation
              (final ActionMapping mapping, final String name){
       // .....
205      if (exclamation != -1) {
206          mapping.setName(name.substring(0, exclamation));
207          if (this.allowDynamicMethodCalls) {
208              mapping.setMethod(name.substring(exclamation + 1));
209          }
210          else {
211              mapping.setMethod((String) null);
       // .....
214  }
```

(b) **A vulnerable target API**(in struts2-rest-plugin 2.3.24.1)

Fig. 3: Example 2 of vulnerability-fixing patch and a target API that is vulnerable. Highlighting the importance of variable propagation in identifying unpatched APIs

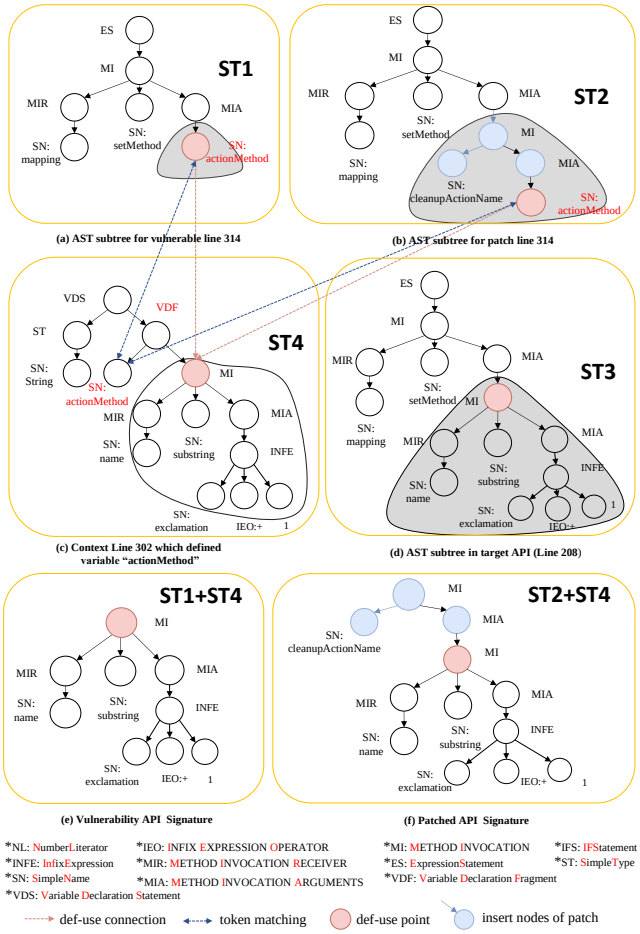


Fig. 4: Solutions of Example 2 through vulnerable trace refinement, highlighting variable propagation is critical for accurately identifying vulnerabilities.

in each project that includes them, which implies that merely identifying the inclusion of a vulnerable library is insufficient. Some recent studies propose leveraging call graph analysis to check whether vulnerable TPL APIs [1], [4], [9] are reachable from a given application project. However, the true challenge lies not just in determining the reachability of vulnerable TPL APIs, but in accurately identifying whether these reachable APIs are still unpatched and actively used within an application, as these pose the most immediate threats. To overcome the challenge, these

studies rely on metadata to assess whether the reachable TPL APIs are vulnerable or have been patched. Typically, these methods collect vulnerabilities in a TPL by searching public vulnerability databases, such as the *Snyk* [23], and then crawling metadata. They subsequently analyze whether the vulnerable API paths are reachable, and finally, during detection, they match part of the metadata, such as by performing string matching with the version number of the TPL used in the application project to determine if it corresponds to a vulnerable version. Metadata plays a critical role in this process, as it encompasses details such as the library name, affected versions, and vulnerability descriptions [9], directly impacting the ability to identify truly at-risk components. Unfortunately, metadata may not always be complete or consistent, leading to significant gaps in vulnerability identification [1], [9]. This issue arises from several key factors: first, the quantity of metadata available for TPL versions is often limited, typically only covering officially released versions; second, the names of TPLs can be easily modified, particularly when developers engage in secondary development, repackaging, or partial repackaging, which makes it difficult to match the modified libraries with the original metadata.

This gap underscores the need for robust techniques to identify unpatched APIs specifically. These are the components that, if left unaddressed, pose the most significant risks to application security. Therefore, the task of identifying unpatched APIs is crucial. Instead of relying on metadata, we leverage a pre-constructed vulnerability database to determine whether a given TPL API is vulnerable or has been effectively patched, based on known vulnerabilities and their corresponding patches.

2.2 Limitations of Existing Approaches

The identification of unpatched APIs typically involves matching a target API's source code against known vulnerabilities and their fixes, encompassing two main methodological approaches: clone-based and patch-enhanced matching [10], [11], [12], [13], [14], [15], [16], [17].

Clone-based approaches [12], [15], [16], [17], [18], [19] extract signatures from both vulnerable and patched APIs, comparing these against a target API to assess similarity. If a target API resembles the signature of a vulnerable API more than its patched version, it is deemed vulnerable. These methods use various signature extraction techniques, including text-based [18], token-based [15], [16], tree-based [19], and graph-based [12], [17] representations. Specially, Tree-based techniques utilize ASTs to identify structural nuances in code, which remain consistent despite superficial changes like whitespace [19]. Graph-based methods consider both syntax and semantics, constructing Program Dependency Graphs (PDG) that combine data flow and control flow graphs to create semantic signatures that reflect code's functional aspects more intricately [12], [17]. These methods typically describe entire code fragments using ASTs or PDGs but essentially focus on detecting overall similarity between code sections. Owing to the inherent nature of clone detection and a lack of consideration for how vulnerabilities are specifically addressed, these approaches struggle to differentiate subtle differences between vulnerable and patched APIs, often leading to high rates of false positives [20].

Patch-enhanced matching approaches [10], [11], [14], [20], [21], [22] incorporate information from vulnerability patches, unlike traditional clone-based approaches, which primarily focus on the unchanged parts of the code. These patch-enhanced methods specifically target the code modifications introduced in patches to create more precise API signatures. For example, MVP [20] utilizes program slicing techniques to extract statements related to vulnerability patches and generate function signatures. Similarly, MOVERY [21] employs function collation and core line extraction to focus on essential code lines, creating signatures that summarize key changes during updates. SECURESYNC [14] introduces an enhanced AST to generate vulnerability patch signatures, using feature vectors to calculate the textual and structural similarities of code trees. VISION [22], on the other hand, focuses on the affected library versions by identifying critical methods and critical statements to generate vulnerability and patch signatures, and calculates semantic similarity using pre-trained models. Despite these techniques using different forms to measure critical vulnerable code lines, such methods largely maintain coarse-grained representations at the statement level, relying on simple algorithms, such as the MD5 hash algorithm, to convert statements into signatures. This coarse-grained approach often struggles to distinguish between critical elements, such as those directly related to vulnerabilities or patches, and non-critical elements within the statements, thereby reducing the effectiveness in accurately identifying nuanced vulnerabilities. For example, using a statement-level signature might lead to mismatches such as the modified line (Line 156) in Figure 1(a) not aligning with Line 136 in Figure 1(b), resulting in a false negative. This motivates the need for a more fine-grained approach to differentiate critical vulnerable elements like "`backendManager.isCorsAccessAllowed`" from non-critical elements such as "`pOrigin!=null`".

2.3 Our Solution

To precisely target critical vulnerable elements within patches, we propose a fine-grained methodology - *patch-enhanced AST-level signature*, which provides a more accurate localization of vulnerability modifications. The patch-enhanced AST-level signature is realized through two key steps.

The first step is *Patch-Induced AST Difference Extraction*. Motivated by tree-based clone-based approaches, we leverage the AST to more precisely describe the structure and characteristics of the code, with the goal of localizing critical vulnerable elements. This step involves extracting subtrees related to changes made during the vulnerability fix, enabling the pinpointing of the critical vulnerable element. For instance, by applying our method to the vulnerability-fixing patch in Figure 1(a) and the target API in Figure 1(b), we are able to identify the key vulnerable elements of the *vulnerable line* and the *patch line*, highlighted in gray in Figure 2. We observe that the AST subtree of the target API (Figure 2(c)) closely matches the AST subtree of the patched version of the API (Figure 2(a)).

The second step is *Vulnerability Trace Refinement*. Based on our observations, the genesis of vulnerabilities is frequently linked to variable propagation. Figures 3 and 4 present

another example along with its corresponding solution, highlighting that variable propagation is critical for accurately identifying vulnerabilities. Figure 4(a)(b) illustrates the first step in extracting the differential AST subtrees ST_1 and ST_2 as vulnerability and patch signatures, respectively. However, if we solely rely on the patch-induced AST difference extraction method and use the differential subtrees ST_1 and ST_2 as signatures to match the target API, it remains unclear whether ST_3 in Figure 4(d) actually correspond to ST_1 or ST_2. By carefully examining the code, we find that Line 208 in Figure 3(b) is equivalent to the previously deleted Line 314 in Figure 3(a) , since the expression "`name.substring(exclamation+1)`" at Line 208 in Figure 3(b) matches the variable initialization "`actionMethod = name.substring(exclamation+1)`" at Line 302 in Figure 3(a). Without this semantic information, it is challenging to determine which subtree of the target API should be used for matching. This step refines the extracted subtrees by tracking variable interactions through program slicing, seamlessly integrating context-specific AST segments to form a complete subtree that effectively highlights critical changes. Unlike traditional PDG methods, we focus solely on tracking variable propagation, which increases analysis efficiency. In contrast to CLDIFF's code difference linking, which establishes relationships between simplified code difference statements using predefined links [29], our approach focuses on leveraging AST stitching to combine the tracked variables into a complete difference subtree. This method does not rely on using full semantic information as API signatures, allowing for a more targeted and efficient analysis. The refined differential subtrees then serve as precise, enhanced signatures, significantly improving the accuracy of vulnerability detection. In Figure 4(a)(b), the code at Line 314 involves the variable `actionMethod`. Through program slicing, we can trace the definition of `actionMethod` at Line 302, then represent it using the corresponding AST in Figure 4(c) as subtree ST_4. By employing the def-use relationships, we stitch ST_1 and ST_2 together with ST_4, as shown with the red dashed line, creating a fine-grained signature shown in in Figure 4(e)(f) that enables matching the otherwise hard-to-locate ST_3 to its similar part ($ST_1 + ST_4$), thus detecting the vulnerability.

In summary, our fine-grained *patch-enhanced AST-level signature* method to precisely identifies critical vulnerable elements, significantly enhancing our ability to detect unpatched APIs with improved accuracy and efficiency.

3 METHODOLOGY

In this section, we present the overview of PATEN and provide some critical definitions. Key steps include extracting a patch-enhanced AST-level signature and employing a similarity algorithm for feature matching to identify unpatched APIs.

3.1 Overview and Definitions

PATEN is designed to ascertain whether a given TPL API is vulnerable or has been patched, taking into account specific known vulnerabilities and patches. Figure 5 shows the overview of PATEN. We generate an AST-level signature for the target API and generate a AST-level vulnerability and patched API signature for the vulnerability-fixing patch (see Section 3.2). Then, we apply a tree similarity algorithm to determine which is a closer match between the target API and the vulnerabilities and patches. (see Section 3.3)

We introduces the key definitions used in our approach. In the remaining of this paper, we assume that each vulnerability is within one API. We first define the AST-level signature as follows.

Definition 1. (AST-level Signature) An AST-level API Signature (SIG), fundamentally represents the AST (FT) of the complete API code, encapsulating its structural composition and illustrating the hierarchical organization of its programming constructs. An AST includes the following key elements:

- **Subtree (ST):** A subtree in the AST represents a coherent subset of the program. Subtrees are critical for analyzing parts of the code related to specific functionalities or modifications.
- **Node (SN):** Each node in the AST represents a specific programming construct such as a statement, expression, or declaration. Each SN is also considered a special type of ST, specifically a ST that contains only one node, thus representing the simplest form of a subtree.

Before defining the patch-enhanced AST-level signature, we present a formal definition of the AST difference. Given a vulnerable API, denoted as API_v, and a patched API, denoted as API_p, we parse them into their respective ASTs, denoted as FT_v and FT_p. By comparing FT_v with FT_p, we can identify the difference subtree, which represents the modifications made in API_p to address the vulnerabilities found in API_v.

Definition 2. (AST Difference) We define the AST Difference as a tuple of two sets.

$$AST_{\text{difference}} = (DSet_{add}, DSet_{del})$$

These sets represent the differences between the ASTs of a vulnerable API and its patched version. More formally, we define the elements of the two lists as follows.

$$DSet_{add} = \{ST \mid ST \in FT_p \wedge ST \notin FT_v\}$$

$$DSet_{del} = \{ST \mid ST \in FT_v \wedge ST \notin FT_p\}$$

With the formal representation of AST difference established, we now define the patch-enhanced AST-level signature.

Definition 3. (Patch-enhanced AST-level Signature) A patch-enhanced AST-level signature ($PSIG$) is a set wherein each element is a subtree (ST), and each subtree represents the modifications made in the patched API (API_p) to address vulnerabilities found in the vulnerable API (API_v). The patch-enhanced AST-level signature is subdivided into: vulnerability API signature ($PSIG_v$) and patched API signature ($PSIG_p$). Specifically, $PSIG_v$ and $PSIG_p$ are subsets of $DSet_{del}$ and $DSet_{add}$, respectively. More examples and illustrations will be elaborated in Section 3.2.

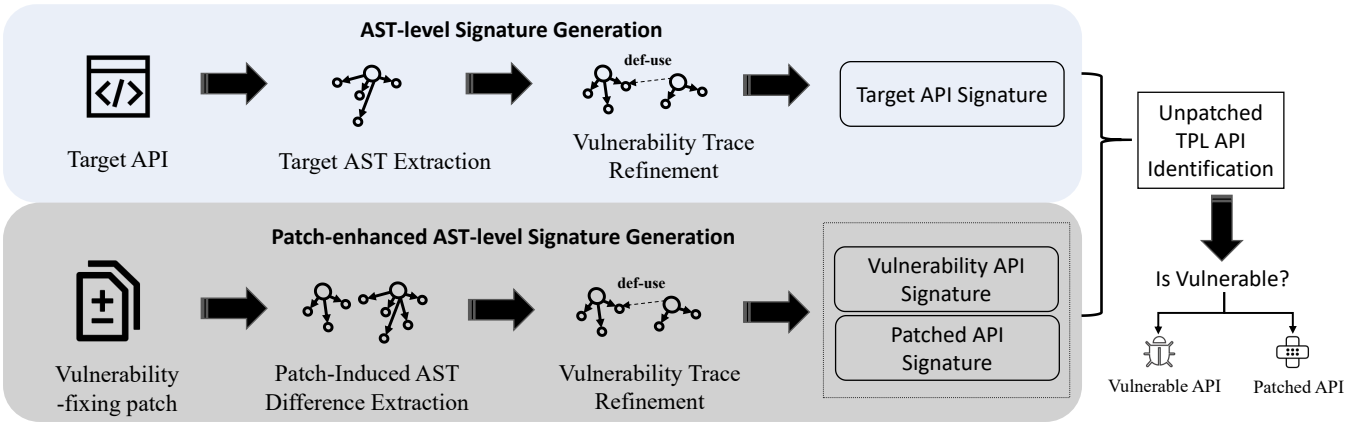


Fig. 5: Approach overview of PATEN

Algorithm 1: Patch-induced AST Difference Extraction

Require: Vulnerable and patched API AST: FT_v, FT_p
Ensure: Patch-induced AST difference set: $DIFF_v$, $DIFF_p$
function ExtractDiff(FT_v, FT_p)
  $DSet_{add} \leftarrow \emptyset$, $DSet_{del} \leftarrow \emptyset$
  for each $ST \in FT_p$ **do** // construct $DSet_{add}$
    if $ST \notin FT_v$ **then**
      $DSet_{add} \cup \{ST\}$
    end if
  end for
  for each $ST \in FT_v$ **do** // construct $DSet_{del}$
    if $ST \notin FT_p$ **then**
      $DSet_{del} \cup \{ST\}$
    end if
  end for
  $DIFF_v \leftarrow$ Extract common ancestors for subtrees in $DSet_{del}$
  $DIFF_p \leftarrow$ Extract common ancestors for subtrees in $DSet_{add}$
  return $DIFF_v$, $DIFF_p$

Algorithm 2: Vulnerability Trace Refinement

Require: Patch-induced AST difference set: $DIFF_v$, $DIFF_p$
Require: Vulnerable and patched API AST: API_v, API_p
Ensure: Patch-enhanced AST-level signatures: $PSIG_p$, $PSIG_v$
function RefineTraces($DIFF, API$)
  $PSIG \leftarrow \emptyset$
  for each $diff \in DIFF$ **do**
    for each $node$ **in** $diff$ **do**
      if $node$ is variable **then**
        $res \leftarrow$ slicing($node$, API)
        if res affects vulnerability context **then**
          $PSIG \cup \{connect(diff, res)\}$
        end if
      end if
    end for
  end for
  return $PSIG$
end function
$PSIG_v \leftarrow$ RefineTraces($DIFF_v, API_v$)
$PSIG_p \leftarrow$ RefineTraces($DIFF_p, API_p$)
return $PSIG_v$, $PSIG_p$

3.2 Patch-enhanced AST-level Signature Generation

To locate the critical elements of vulnerabilities in a more fine-grained level, we propose the patch-enhanced AST-level signature, which includes two main steps: patch-induced AST difference extraction and vulnerability trace refinement.

3.2.1 Patch-induced AST Difference Extraction.

Patch-induced AST difference extraction process involves analyzing the structural changes in the ASTs of a vulnerable and patched API. This process helps to identify specific modifications that fix vulnerabilities.

Algorithm 1 outlines the extraction process for identifying changes between the ASTs of vulnerable and patched APIs. From these ASTs, differential subtrees are compiled into two distinct sets: $DSet_{add}$ for additions and $DSet_{del}$ for deletions. For example, the subtree *SN:isCorsAccessAllowed* shown in Figure 2(a) is cataloged under $DSet_{del}$, while the subtrees *SN:isOriginAllowed* and *BL:false* from Figure 2(b) are listed under $DSet_{add}$. Although the differential subtrees in either $DSet_{del}$ or $DSet_{add}$ are finer granularity than statement, a single differential subtree provides limited information for matching. Thus, we propose to aggregate the differential subtrees of the same statement together by the algorithm of finding the lowest common ancestors in the AST [30]. For example, the two subtrees *SN:isOriginAllowed* and *BL:false* in Figure 2(b) share the lowest common ancestor *MI*, which is a more syntactically complete and representative subtree for matching. Therefore, we select the subtree rooted at *MI* as the aggregated differential subtree. Note that our aggregation is performed only at the statement level of AST subtrees to preserve the merits of fine granularity.

3.2.2 Vulnerability Trace Refinement

As mentioned in Section 2.3, the genesis of vulnerabilities is often closely associated with variable propagation within the code structure. This refinement phase is designed to increase the accuracy of vulnerability identification by directly correlating code modifications to the contexts of vulnerabilities.

Algorithm 2 shows the process of vulnerability trace refinement. For each node in the differential subtrees, if the node represents a variable, we perform program slicing on it. This technique allows for the inclusion of relevant contextual statements that aid in more precise vulnerability matching. We perform forward and backward slicing on API_v (resp. API_p) using the variables appearing at the deleted statements (resp. the added statements) as the selection criterion, but only focus on computing those statements that have a def-use relationship on the selected variables. To avoid introducing too many statements where some of them are

irrelevant to the vulnerability, we only slice the statements that directly define or use the selected variables. The result of the slicing is then transformed back into the AST format. These contextual AST segments, once integrated with the differential ASTs through def-use edges, effectively form connections that serve as the final patch-enhanced AST-level signature.

For example, as shown in Figure 3(a), our approach includes only Line 302 as the contextual statement for analysis. In Figure 4, we establish def-use edges from *SN:actionMethod* in both ST_1 and ST_2 to the method invocation node *MI*. Subsequently, ST_1 (respectively, ST_2), which is connected to ST_4 through these def-use edges, serves as the final signature for identifying vulnerabilities (respectively, patches).

3.3 Unpatched API Identification

To identify unpatched APIs, we construct both a target API signature and a vulnerability and patched API signature. The method for constructing the target API signature is similar to the patch-enhanced AST-level signature generation process. Firstly, we parse the target API into an AST. To ensure alignment with the signatures for vulnerabilities and patches, we conduct vulnerability trace refinement. This refinement culminates in the formation of the final target API signature.

To determine which is a closer match between the target API and the vulnerabilities and patches, we employ tree edit distance [31] as a measure of similarity. This choice is predicated on the ability of tree edit distance to quantitatively reflect the minimal changes needed to transform one tree into another, providing a nuanced and precise comparison of AST structures [32], [33]. This metric is particularly effective in capturing the structural nuances essential for accurate API signature comparisons.

Current tree edit distance algorithms do not consider node weights [31]. To address this limitation and incorporate vulnerability characteristics, we introduce a node weighting mechanism and a cost function.

Definition 4. (Cost Function) We first define the weight for each node.

$$Weight(SN) = |Numof_v(SN) - Numof_p(SN)|$$

Where $Numof_v(SN)$ represents the frequency of the node in the vulnerable API AST(FT_v) and $Numof_p(SN)$ represents its frequency in the patched API AST(FT_p).

$$Cost(\text{add}) = Cost(\text{del}) = Weight(SN) \times Cost_{\text{op}}$$

$Cost_{\text{op}}$ represents the base operation cost for adding or deleting a node, which is typically standardized across all nodes. The $Weight(SN)$ adjusts the cost of the operation based on the frequency difference of the node between the vulnerable and patched API ASTs.

To be more specific, given two trees T_1 and T_2, with the definition of a cost function for the tree edit operations (including node deletion and addition), the minimum cost to convert from T_1 to T_2 is the edit distance, denoted as $dist(T_1, T_2)$. For example, in Figure 4, where the weight of all nodes is set to 1, to convert from the combination tree

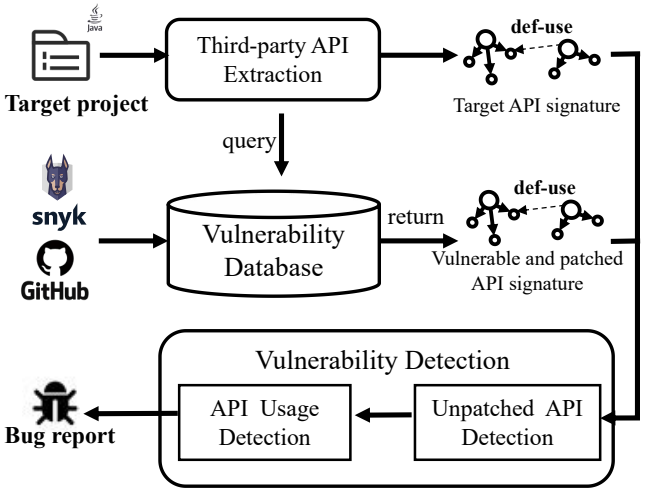


Fig. 6: Implement architecture of PATEN

of $ST_2 + ST_4$ to the tree ST_3, we only need to conduct three addition operation on the node *SN:cleanActionName*, *MI* and *MIA*, and thus the edit distance $dist(ST_2 + ST_4, ST_3)$ is three. Similarly, the combination tree of $ST_1 + ST_4$ and the tree ST_3 are identical, meaning that the edit distance $dist(ST_1 + ST_4, ST_3)$ is zero.

Intuitively, the value of the edit distance can be very large if two comparing trees are very large. To mitigate the impact of the scale of trees, we normalize the edit distance with the total number of nodes from the two given trees. More formally, the similarity metric is defined as follows.

$$Sim(T1, T2) = \frac{dist(T_1, T_2)}{NumOfNodes(T_1) + NumOfNodes(T_2)}$$

where $NumOfNodes(T_1)$ and $NumOfNodes(T_1)$ represent the number of nodes of T_1 and T_2. By comparing the values of $Sim(SIG, PSIG_v)$ and $Sim(SIG, PSIG_p)$, we determine which version of the API, the one before or after the vulnerability fix, is more similar to the target API. However, similar to previous studies [14], [20], [21], [22], merely comparing the relative similarity is insufficient to definitively identify the presence of a vulnerability. When both similarity values between the target API and the vulnerability-related APIs are below a certain threshold, we no longer rely on relative distance comparisons. We introduce a similarity threshold t to filter out APIs unrelated to the vulnerability. When $Sim(SIG, PSIG_v) < t$ and $Sim(SIG, PSIG_p) < t$, the target API can be considered unrelated to the vulnerability. We selected nine threshold values from 0.1 to 0.9 with a precision of 0.1 and validated them on *PATEN-BENCH* shown in Table 1. The experimental results show that the best matching performance is achieved when the similarity threshold is set to 0.5.

4 Implementation

We implement PATEN to identify real-world unpatched TPL APIs, comprising a total of 27,683 lines of code. This paper primarily focuses on the detection of unpatched TPL APIs in Java. In Section 5.6, we will discuss in detail the

reasons and the extensibility of the proposed method. Figure 6 illustrates the architecture of the tool. The process begins with the extraction of third-party APIs from the target project. These APIs are then cross-referenced against the vulnerability database to match their fully qualified names. If a match is found, a target API signature is generated. Subsequently, unpatched API detection and API usage detection are conducted using the generated signatures, leading to the generation of a bug report if vulnerable APIs are found to be reachable within the application. Below, we detail the key aspects of its implementation.

Vulnerability Database Construction. We construct a vulnerability database for PATEN that includes signatures for known vulnerabilities and their patches. Access to comprehensive data on vulnerability fixes is crucial as the quality and scope of this dataset directly affect our ability to detect unpatched TPL APIs. To assemble a thorough and reliable vulnerability dataset, we use *Snyk* [23], a leading security platform, in conjunction with GitHub [24]. *Snyk* is preferred over traditional vulnerability databases such as CVE and NVD due to its integration of data from multiple sources, which not only simplifies our data collection process but also provides broader vulnerability coverage. Furthermore, our interactions with the developers at *Snyk* have assured us of the data's integrity; it is rigorously analyzed, tested, and enriched before inclusion, ensuring high quality and reliability.

To systematically identify vulnerabilities and their corresponding patches, we have automated our data collection process as follows:

- We extracted Java vulnerabilities from the *Snyk* database using the "Maven" filter.
- We confirmed the existence of patches (e.g., git commits, pull requests) for each vulnerability and ensured the source code was hosted on GitHub, recording relevant URLs.
- Using `git clone` and `git diff`, we downloaded and analyzed the modified code to determine which APIs were affected by the patches, utilizing the tool *Understand* [34].

In summary, we collected data on 334 known vulnerabilities along with their corresponding patches, impacting a total of 1,701 APIs. We implemented the patch-enhanced AST-level signature generation on the GUMTREE [35], which is a AST differencing tool. We have open-sourced our vulnerability dataset on GitHub[1].

Third-party API Extraction. We firstly analyze the dependency trees of Java projects using the MAVEN [36] build system, which provides detailed insights into the libraries and their versions used. Next, we employ the Java static analysis tool SOOT [37] to extract the fully qualified names of all APIs from third-party libraries, including API names, parameter lists, parameter types, and return values. We then use the fully qualified names to query a vulnerability database for matching vulnerability-fixing patches, which are used for subsequent vulnerability detection.

Vulnerability Detection. Vulnerability detection is divided into two stages. In the first stage, we follow the most classical tree edit distance algorithm [31] to identify unpatched APIs. In the second stage, our goal is to check whether an unpatched TPL API is reachable from the application project. As pointed out by prior studies [1], [4], this is important for developers to ensure the security impacts of the unpatched TPL APIs. The detailed process for detecting API usage is as follows:

Initially, we verify whether the TPL containing the unpatched API is loaded, by examining the configuration of the build management tool. This step is crucial to address the dependency conflict issues, identified in earlier research [38], where an application may incorporate multiple versions of the same library but only one is actually loaded. We utilize the *Maven Plugin API* [39] to extract the library dependency tree of the application project and dismiss any libraries not loaded by MAVEN. If the TPL of the unpatched API passes this filter, we employ SOOT [37] to construct a call graph for the application project along with all TPLs included from the initial filtering step. Utilizing the CHA mode of Soot, we analyze call relationships across all methods in the TPLs, integrating these into a comprehensive call graph. Beginning from all public methods in the target project, a depth-first search algorithm is executed to find paths that can reach the vulnerable method. If a path is found, we generate a detailed bug report documenting the use of this unpatched TPL API.

5 EVALUATION

Our evaluation is designed to answer the following research questions:

- **RQ1:** How effective and efficient is PATEN at identifying unpatched TPL APIs?
- **RQ2:** Does PATEN effectively detect and help diagnose the use of unpatched TPL APIs in real-world projects?
- **RQ3:** How does patch-induced AST difference extraction in PATEN help detect unpatched APIs?
- **RQ4:** How does vulnerability trace refinement in PATEN help detect unpatched APIs?

5.1 Experimental Setting

Dataset Preparation. Due to the absence of a large-scale dataset, we constructed a comprehensive dataset named *PATEN-DATASET*. This dataset comprises two main components: a vulnerability database and a benchmark dataset named *PATEN-BENCK*, which includes APIs tagged with their vulnerability statuses.

1) **Vulnerability Database Construction:** The vulnerability database includes signatures for both vulnerable and patched TPL APIs. This database aids in accurately assessing whether a given TPL API has been patched, as detailed in Section 4.
2) ***PATEN-BENCH* Construction:** To ensure a robust evaluation of PATEN and avoid the potential for overfitting, we did not use API versions immediately before and after patch applications as part of our test set. Instead, for each TPL version associated with known vulnerabilities from the vulnerability database, we collected all official release versions of these TPLs. We then queried the *Snyk* database to determine if these versions are classified as vulnerable or patched with respect to the known vulnerabilities. The results formed the *PATEN-BENCH*, a comprehensive list of

1. https://github.com/PATEN-Tool/PATEN_vuldb_info

30,389 API versions each labeled as vulnerable or patched, which is detailed in Table 1. Due to space constraints, we have only listed the top 50 third-party libraries, ranked by the number of associated CVEs, in descending order of the number of CVEs per library.

It should be noted that the baselines used in our evaluation are not dependent on specific dataset sizes or sets of vulnerabilities, and they do not rely on any imbalanced signature distribution. REDEBUG, VUDDY, MVP, and MOVERY collected C++ vulnerabilities and security patches from the National Vulnerability Database (NVD) [40] as their vulnerability datasets. Due to confidentiality, their vulnerability datasets have not been publicly released. On the other hand, VISION collected 12,073 Java library versions corresponding to 102 CVEs across 79 libraries from the NVD [22]. It is important to highlight that the CVEs and the number of versions in our dataset are more than twice that of VISION, and our dataset nearly covers all of the libraries used in VISION.

TABLE 1: The number of patched and unpatched TPLs used in Github projects.

No.	Vulnerable TPL	Star/Forks	#vulnerable	#patched	#total
1	tomcat-embed-core	7.4k/5k	1,129	3,530	4,659
2	tomcat-catalina	7.4k/5k	1,461	2,090	3,551
3	spring-oxm	56k/37.9k	329	2,062	2,391
4	tomcat-coyote	7.4k/5k	393	1,661	2,054
5	spring-web	56k/37.9k	249	979	1,228
6	keycloak-services	21.8k/6.6k	429	297	726
7	elasticsearch	69k/24.5k	371	255	626
8	postgresql	1.5k/835	314	285	599
9	undertow-core	3.6k/986	438	110	548
10	spring-security-web	8.7k/5.8k	194	329	523
11	activemq-all	2.3k/1.4k	112	396	508
12	solr-core	1.1k/636	348	159	507
13	ognl	216/77	35	360	395
14	activemq-broker	2.3k/1.4k	22	334	356
15	keycloak-model-jpa	21.8k/6.6k	193	144	337
16	activemq-client	2.3k/1.4k	78	259	337
17	spring-security-core	8.7k/5.8k	105	222	327
18	jetty-servlets	3.8k/1.9k	305	22	327
19	spring-messaging	56k/37.9k	93	227	320
20	phoenix-core	1k/998	87	168	255
21	spring-data-commons	766/664	85	166	251
22	async-http-client	6.3k/1.6k	241	0	241
23	rabbitmq-jms	62/49	74	159	233
24	hawtio-system	1.4k/540	145	88	233
25	hadoop-common	14.6k/8.8k	50	176	226
26	tomcat-catalina-jmx-remote	7.4k/5k	114	92	206
27	nifi-web-security	4.7k/2.7k	154	42	196
28	activemq-jaas	2.3k/1.4k	20	168	188
29	jruby-core	3.8k/922	12	173	185
30	resteasy-jaxrs	2/0	0	184	184
31	spring-security-crypto	8.7k/5.8k	96	80	176
32	vaadin-server	1.8k/729	90	74	164
33	artemis-openwire-protocol	932/917	152	12	164
34	spring-core	56k/37.9k	27	134	161
35	infinispan-server-rest	1.2k/628	91	70	161
36	orientdb-server	4.7k/869	41	116	157
37	jetty-http	3.8k/1.9k	12	144	156
38	jooby	1.7k/200	128	27	155
39	opencast-caption-impl	383/231	130	20	150
40	vertx-core	14.2k/2.1k	24	120	144
41	hive-exec	5.5k/4.7k	39	99	138
42	activemq-osgi	2.3k/1.4k	9	126	135
43	uimaj-core	63/37	70	63	133
44	jolokia-core	808/218	98	33	131
45	weld-core	382/285	129	0	129
46	weld-core-impl	382/285	128	0	128
47	struts2-core	1.3k/810	38	89	127
48	concord-server-impl	207/102	0	126	126
49	spring-cloud-dataflow-server-core	1.1k/578	105	21	126
50	coyote	7.4k/5k	50	76	126
51	**others**		2,599	2,186	4,785
Total			11,636	18,753	30,389

#vulnerable: the number of API versions that are vulnerable.
#patched: the number of API versions that are patched.
#total: the total number of API versions.

Baselines. We compare PATEN with six baselines based on patch-enhanced matching approaches.

- REDEBUG [10] uses context-based code matching by combining pre-patch and post-patch code snippets to identify vulnerabilities.
- VUDDY [11] matches API signatures directly by replacing and normalizing function signatures to detect vulnerabilities.
- MVP [20] utilizes program slicing techniques to extract relevant code related to vulnerabilities and compares them with the target API.
- MOVERY [21] focuses on core code lines and function collation to summarize key changes during updates, creating concise vulnerability signatures.
- SECURESYNC [14] utilizes an enhanced AST with labels to compare structural features of vulnerable code snippets, providing reliable support for detecting recurring vulnerabilities.
- VISION [22] prioritizes key changes and their contexts through critical method selection and critical statement identification, enabling accurate identification of the affected library versions for vulnerabilities.

We clarify that the underlying principles of these baselines are language-agnostic. These methods aim to generate vulnerability patch signatures through program analysis techniques and are broadly applicable to the detection of Java unpatched TPL APIs. However, to ensure fairness, we also carefully considered the specific languages supported by the implementations of these tools. REDEBUG is applicable to Java, so we directly used its source code. VISION is designed to detect vulnerable affected TPL versions in Java. Since our focus is on Java unpatched APIs, we set the critical methods in VISION as the TPL APIs we aim to detect. VUDDY and MOVERY are designed for C++ code, so we modified them to make them compatible with Java. Since MVP is a proprietary tool, its source code is not available. In our experiments, we endeavored to implement and configure it based on the descriptions provided in its original paper specifically for Java programs. We engaged in multiple communications with the authors of MVP to ensure that our implementation accurately reflected the details described in the paper, and our tests confirmed that it achieved the expected results. SECURESYNC's source code is also not available. We contacted the authors to inquire whether they could share the source code, but we did not receive a response. As a result, we replicated it for Java based on the details described in the paper. To verify the correctness of our implementation, we manually wrote test cases and validated the output.

We do not compare PATEN with clone-based approaches due to the significant limitations highlighted by MVP [20]. According to their findings, clone-based methods such as SOURCERERCC [15] and CCALIGNER [16] exhibit extremely low precision (0.5% and 0.3%, respectively) and moderate recall (64.9% and 56.8%, respectively) when applied to the task of identifying vulnerable functions. This demonstrates that clone-based approaches are not well-suited for this type of task, as they produce a large number of false positives.

Metrics. We employ several standard metrics to assess the performance of PATEN including precision, recall, accuracy,

and F1-measure. We first define the basic metrics as follows:

- True Positive (TP): the API identified as vulnerable is actually vulnerable.
- False Positive (FP): the API identified as vulnerable but is actually patched.
- True Negative (TN): the API identified as patched is indeed patched.
- False Negative (FN): the API identified as patched but is actually vulnerable.

Based on these basic metrics, we calculate the following performance metrics: recall, precision, accuracy, and F1-measure. Precision assesses the accuracy with which PATEN detects vulnerable APIs. Recall measures the tool's ability to identify all vulnerable APIs. Accuracy reflects the overall ability to correctly classify both vulnerable and patched APIs. F1-measure provides a harmonic mean of precision and recall.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$F1 - measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

We also utilize the DTime to measure the detection efficiency of PATEN. This metric represents the total time taken to detect whether an API is unpatched including both analysis time (the time required to generate the target API signature and the vulnerability/patch API) and matching time (the time spent searching for vulnerabilities in target APIs).

Environment. We ran our experiments on a server with one hundred and four "Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz" processors and 512GB of memory running Ubuntu 20.04.6 LTS.

5.2 Effectiveness and Efficiency

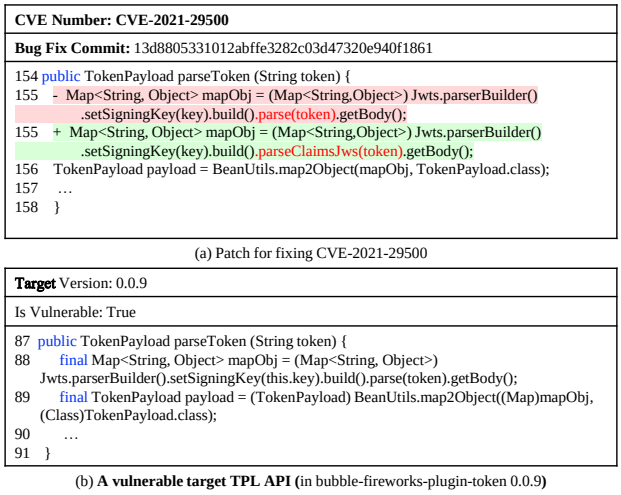
We applied PATEN and the baselines to identify unpatched TPLs in *PATEN-BENCK*. The results are demonstrated in Table 2.

Effectiveness. As shown in Table 2, PATEN demonstrates strong effectiveness, achieving a high precision of 90.41%, a recall of 93.96%, an accuracy of 93.87% and an F1-measure of 92.15%. These results indicate that PATEN not only detects unpatched TPL APIs with high accuracy but also maintains a good balance between precision and recall. In contrast, while some baseline approaches, such as VUDDY, REDEBUG, MVP, and VISION, achieve a perfect precision of 100%, this comes at the expense of significantly lower recall. The low recall rates lead to much lower F1-measure and accuracy scores, suggesting that these baselines fail to detect a significant portion of unpatched TPL APIs that our approach successfully identifies. The poor recall performance of the baseline approaches can be attributed to their reliance on statement-level signatures, which fail to capture the critical vulnerable elements within a statement. As a result, mismatches frequently occur between the target API signature and the vulnerable API signature, limiting their ability to identify vulnerabilities. Figure 7 shows a concrete example where all baseline approaches miss this vulnerable API, whereas PATEN successfully identifies it. All baseline approaches, which rely on coarse-grained representations, fail to match the modified line (Line 155 in Figure 7(a)) with Line 88 in Figure 7(b), leading to a false negative. VUDDY, REDEBUG, MVP, and MOVERY use hash values of code statements as signatures. Hash values only match exactly when the textual content is identical, which leads to mismatches due to non-critical elements, such as the "`final`" keyword, in the target API's vulnerable code lines and the vulnerability patch. This results in false negatives. VISION generates code semantic embeddings using UniXcoder and computes the cosine similarity between the target API and the vulnerability-related API. On the other hand, SECURESYNC uses an enhanced AST to generate vulnerability patch signatures, applying feature vectors to assess textual and structural similarities in code trees. Both of these methods, although utilizing different forms of feature representation, rely on coarse-grained representations at the statement level. They fail to eliminate the influence of non-critical elements like "`final`", leading to false negatives. In the target API, such non-critical elements related to vulnerabilities and patches are very common in Java APIs. Changes in the target API code lead to code fragments that are not exactly similar to those before and after the vulnerability fix. This type of API is present in *PATEN-BENCH* at a rate of 39%. In contrast, PATEN adopts fine-grained AST-level signatures, enabling the precise identification of critical vulnerable elements , as shown in Figure 7(a) with the example of "`parse(token)`". This fine-grained approach ensures that PATEN can effectively detect unpatched TPL APIs that are missed by statement-level signature-based methods. When comparing the overall performance, PATEN clearly outperforms the baselines. In terms of accuracy, PATEN surpasses VUDDY, REDEBUG, MVP, MOVERY, VISION, and SECURESYNC by 51.3%, 51.4%, 49.0%, 51.9%, 9.7%, and 17.9%, respectively. Similarly, in terms of F1-measure, PATEN outperforms these baselines by 5,321%, 3,364%, 1,300%, 3,586%, 20%, and 39%, respectively. This significant improvement highlights the effectiveness of our approach in identifying unpatched TPL APIs.

Efficiency. As shown by the AvgDTime column in Table 2, the average detection time per API using PATEN is 796ms. Although PATEN is not as efficient as some tools such as VUDDY (617ms) and REDEBUG (609ms) in terms of detection time, this is largely due to its use of fine-grained AST-level signatures. PATEN performs tree edit distance calculations to determine the closest match between the target API and the vulnerabilities or patches, which incurs additional computational overhead. Among the baselines, VISION demonstrates the highest detection time at 196,830ms. This is because it not only employs a pre-trained model, UniXcoder [41], to generate semantic embedding vectors for each statement but also generates a PDG using Joern [42]. The forward and backward slicing on the PDG, based on changed statements, further increases the detection time. Despite requiring more time compared to certain baselines, PATEN achieves a favorable trade-off between effectiveness and efficiency. Its significantly higher Recall and F1-measure

TABLE 2: Benchmark results of PATEN and baselines.

Tool	TP	FP	TN	FN	Precision	Recall	Accuracy	F1-measure	AvgDTime(ms)
PATEN	10,933	1,160	17,593	703	90.41%	93.96%	93.87%	92.15%	796
VUDDY	100	0	18,753	11,536	100%	0.86%	62.04%	1.70%	617
REDEBUG	157	0	18,753	11,479	100%	1.35%	61.99%	2.66%	609
MVP	396	0	18,753	11,240	100%	3.40%	63.01%	6.58%	610
MOVERY	149	124	18,629	11,487	54.58%	1.28%	61.79%	2.50%	564
VISION	7,251	0	18,753	4,385	100%	62.32%	85.57%	76.78%	196,830
SECURESYNC	6,124	684	18,069	5,512	89.95%	52.63%	79.61%	66.41%	139



(a) Patch for fixing CVE-2021-29500

(b) **A vulnerable target TPL API (**in bubble-fireworks-plugin-token 0.0.9**)**

Fig. 7: An example of vulnerability-fixing patch and a target API that is vulnerable. While all baseline approaches fail to detect the vulnerability, PATEN successfully identifies it.

compared to other tools demonstrate its superior ability to detect unpatched TPL APIs. Furthermore, the detection time of PATEN remains within a reasonable overhead, making it suitable for integration into IDEs for real-time vulnerability detection.

Answer to RQ1: PATEN is highly effective at identifying unpatched TPL APIs, outperforming the baseline approaches in terms of recall and F1-measure. Moreover, the detection time remains within a reasonable range, highlighting its potential for integration into IDEs for real-time vulnerability detection.

5.3 Usefulness

To evaluate the usefulness of PATEN, we employed it to detect unpatched APIs in real-world scenarios. We applied PATEN to approximately 1,500 Maven projects from GitHub, selecting active projects with at least 20 forks or stars and recent updates within the last three months. Table 3 presents a summary of our efforts to assess the impact of PATEN in real-world scenarios. In total, we detect and submit 82 bug reports that are related to the usage of unpatched TPL APIs, of which 71(86.59%) are fixed by developers and 11(13.41%) are already confirmed.

Our analysis of the detected vulnerabilities highlighted that many were addressed in newer versions of the corresponding TPLs. However, developers often persisted in using older versions either due to unawareness of the vulnerabilities or neglect of the updates in the TPLs they incorporated into their projects. Furthermore, some projects indirectly depend on vulnerable TPLs, complicating developers' ability to identify the security risks. This oversight frequently leads to the inadvertent introduction of vulnerabilities into projects when using TPLs.

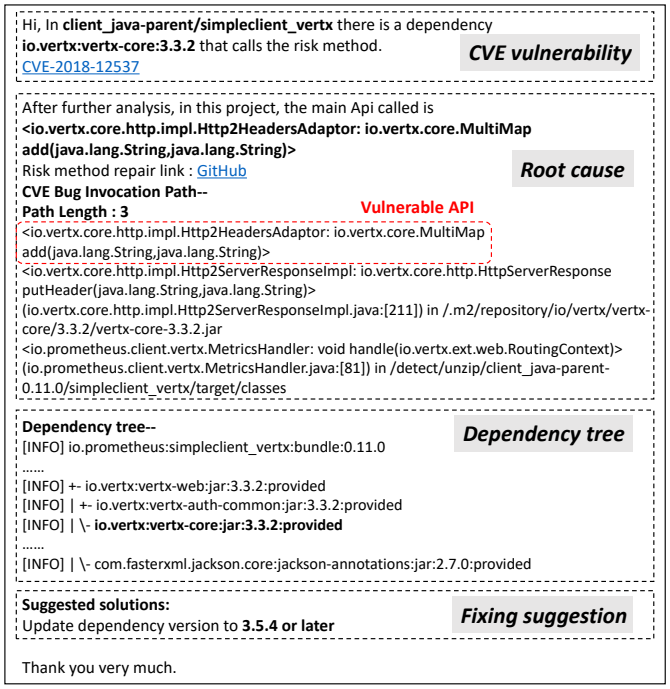


Fig. 8: An example of our detailed bug report.

These bugs reported were swiftly confirmed or fixed, attributed to our approach's fine-grained design, which allows for the provision of detailed root causes for each vulnerability. For instance, Figure 8 showcases a bug report generated by PATEN for the project *client-java* [43], illustrating the effectiveness of our method in real-world applications.

Answer to RQ2: PATEN has proven useful in detecting and diagnosing the use of unpatched TPL APIs across various real-world projects, significantly aiding developers in addressing potential security risks.

TABLE 3: Bug Reports: Vulnerable APIs Detected by PATEN

No.	Project Name	Bug ID	Stars/Forks	CVE ID	Status
1	tutorials	#11291	34.2k/53.6k	CVE-2020-26258	Fixed
2	apollo	#4755	28.1k/10.1k	CVE-2022-25857	Confirmed
3	hutool	#2999	26.4k/7.1k	CVE-2022-25857	Fixed
4	litemall	#497	18k/7k	CVE-2018-1000613h	Fixed
5	openapi-generator	#15195	16.6k/5.3k	CVE-2018-12537	Fixed
6	DataX	#1088	13.5k/4.8k	CVE-2020-13956	Fixed
7	deeplearning4j	#9471	13k/4.9k	CVE-2018-11771	Fixed
8	webmagic	#1038	10.8k/4.1k	CVE-2020-13956	Fixed
9	shardingsphere-elasticjob	#2153	7.9k/3.3k	CVE-2020-13956	Fixed
10	shardingsphere-elasticjob	#1993	7.9k/3.3k	CVE-2021-21290	Fixed
11	sofa-jraft	#960	3.2k/1k	CVE-2022-25857	Fixed
12	ShedLock	#738	3k/442	CVE-2020-13956	Fixed
13	best-pay-sdk	#131	2.8k/903	CVE-2020-13956	Fixed
14	webcam-capture	#879	2.1k/1.1k	CVE-2020-13956	Fixed
15	client_java	#705	2k/739	CVE-2018-12537	Fixed
16	servicecomb-pack	#723	1.9k/449	CVE-2021-21290	Fixed
17	docx4j	#477	1.9k/1.2k	CVE-2020-13956	Fixed
18	azure-sdk-for-java	#34162	1.8k/1.7k	CVE-2022-25857	Fixed
19	forest	#54	1.3k/185	CVE-2020-13956	Fixed
20	inlong	#7480	1.2k/420	CVE-2019-17563	Fixed
21	zstack	#1287	1.2k/385	CVE-2018-11771	Fixed
22	xdocreport	#500	1.1k/342	CVE-2019-12415	Fixed
23	datagear	#10	931/275	CVE-2019-12415	Fixed
24	datagear	#23	931/275	CVE-2022-31692	Fixed
25	alexa-skills-kit-sdk-for-java	#297	805/753	CVE-2020-13956	Fixed
26	esper	#251	791/250	CVE-2020-13956	Fixed
27	alibaba-rsocket-broker	#220	722/156	CVE-2022-25857	Fixed
28	mendmix	#17	664/289	CVE-2022-25857	Fixed
29	aem-core-wcm-components	#2358	648/699	CVE-2020-9484	Fixed
30	jinjava	#1049	601/151	CVE-2022-25857	Confirmed
31	infinitest	#351	572/152	CVE-2020-15250	Fixed
32	Web-Karma	#573	557/196	CVE-2020-13956	Fixed
33	vertx-guide-for-java-devs	#95	544/191	CVE-2019-17640	Confirmed
34	aws-athena-query-federation	#829	498/258	CVE-2020-13956	Fixed
35	nutzboot	#236	487/141	CVE-2019-17638	Fixed
36	congomall	#18	442/55	CVE-2022-25857	Fixed
37	cointrader	#164	428/162	CVE-2019-0201	Fixed
38	helix	#1890	423/207	CVE-2020-13956	Fixed
39	incubator-celeborn	#1380	416/169	CVE-2022-25857	Fixed
40	spring-comparing-template-engines	#105	411/113	CVE-2022-25857	Confirmed
41	joyrpc	#20	404/167	CVE-2016-6345	Fixed
42	msf4j	#589	363/357	CVE-2015-7940	Confirmed
43	camellia	#98	356/90	CVE-2018-1000873	Fixed
44	core-geonetwork	#6759	354/452	CVE-2020-13956	Fixed
45	vertx-microservices-workshop	#48	321/200	CVE-2019-17640	Confirmed
46	webapp-runner	#244	321/111	CVE-2020-9484	Fixed
47	boxable	#241	301/137	CVE-2021-27807	Fixed
48	NCM2MP3	#9	278/54	CVE-2022-25845	Fixed
49	OpenAudioMc	#306	267/90	CVE-2022-25857	Fixed
50	tbschedule	#29	253/149	CVE-2019-0201	Fixed
51	rumble	#1226	194/78	CVE-2022-25857	Fixed
52	h2gis	#1338	191/65	CVE-2022-21724	Fixed
53	h2gis	#1338	191/65	CVE-2022-26520	Fixed
54	JedAIToolkit	#63	191/39	CVE-2020-13956	Fixed
55	wms-ruoyi	#1	176/35	CVE-2021-36090	Fixed
56	scblogs	#169	169/50	CVE-2022-25857	Fixed
57	FastBeeIM	#2	141/39	CVE-2022-25857	Fixed
58	micro-integrator	#2457	138/167	CVE-2016-6812	Confirmed
59	herd	#499	134/41	CVE-2017-5645	Fixed
60	deegree3	#1208	127/98	CVE-2020-9484	Fixed
61	extdirectspring	#171	119/62	CVE-2015-5211	Fixed
62	oxalis	#550	105/81	CVE-2020-13956	Fixed
63	wicket-jquery-ui	#339	92/58	CVE-2021-23937	Fixed
64	rocketmq-connect	#434	90/99	CVE-2022-25845	Fixed
65	hdt-java	#137	88/62	CVE-2018-11771	Fixed
66	BlackLab	#419	85/51	CVE-2022-25857	Fixed
67	Stitching	#70	83/61	CVE-2022-25857	Fixed
68	kungfu	#2	81/61	CVE-2022-25845	Fixed
69	Kubernetes_eShop	#14	63/51	CVE-2020-9484	Confirmed
70	californium.tools	#86	59/56	CVE-2017-7656	Fixed
71	Qanary	#154	52/24	CVE-2022-25857	Fixed
72	iBioSim	#620	50/18	CVE-2020-13956	Confirmed
73	red5-plugins	#37	48/77	CVE-2019-0231	Confirmed
74	vertx-kubernetes-workshop	#21	41/29	CVE-2019-10174	Confirmed
75	jdcloud-sdk-java	#274	37/41	CVE-2020-13956	Fixed
76	xlsmapper	#117	30/9	CVE-2019-12415	Fixed
77	alcor	#760	29/34	CVE-2020-13956	Fixed
78	helidon-build-tools	#850	29/34	CVE-2020-13956	Fixed
79	newrelic-unix-monitor	#55	27/18	CVE-2020-13956	Fixed
80	incubator-hugegraph-commons	#109	26/37	CVE-2020-15250	Fixed
81	poc-spring-with-webapp-gestionmateriel	#39	23/38	CVE-2022-25857	Fixed
82	basyx-java-sdk	#276	22/27	CVE-2021-36090	Fixed

5.4 Contribution of Patch-induced AST Difference Extraction

To evaluate the contribution of patch-induced AST difference extraction in enhancing patch-enhanced AST-level signatures, we conducted an experiment comparing PATEN and PATEN-NoPID, which omits the patch-induced AST difference extraction step. PATEN-NoPID directly uses the vulnerability and patch statements as the AST-level signatures without extracting critical vulnerable elements through the difference extraction process. Table 4 shows the improvement between PATEN and PATEN-NoPID in *PATEN-BENCH*. The inclusion of patch-induced AST difference extraction in PATEN results in notable improvements across all key metrics. Specifically, precision decreases by $9.59\%(=(90.41\%-100\%)/100\%)$, recall increases by $44,857.14\%(=(93.96\%-0.21\%)/0.21\%)$, accuracy increases by $51.95\%(=(93.87\%-61.79\%)/61.79\%)$, and F1-measure increases by $21,488.37\%(=(92.15\%-0.43\%)/0.43\%)$.

TABLE 4: Performance improvement between PATEN and PATEN-NoPID.

Tool	PATEN-NoPID	PATEN	Improvement
Precision	100%	90.41%	9.59% ↓
Recall	0.21%	93.96%	44,857.14% ↑
Accuracy	61.79%	93.87%	51.95% ↑
F1-measure	0.43%	92.15%	21,488.37% ↑

PATEN utilizes patch-induced AST difference extraction to effectively capture minor code changes related to vulnerability fixes. This enables a more accurate identification of critical vulnerable elements within patches. By focusing on the differences in AST structure, including changes in node types, variable names, and other key elements, we can more precisely localize vulnerability modifications. This method helps avoid the inclusion of irrelevant code changes and ensures that the extracted patch signatures are directly tied to the vulnerability fix. In contrast, PATEN-NoPID does not leverage this extraction, which may lead to less accurate localization of critical changes. This results in PATEN-NoPID being highly sensitive to irrelevant code changes, significantly reducing recall and F1-measure.

Answer to RQ3: Patch-induced AST difference extraction in PATEN helps detect unpatched APIs by analyzing critical code changes specific to vulnerability fixes, allowing for precise identification of unpatched elements within the target API.

5.5 Contribution of Vulnerability Trace Refinement

To evaluate the contribution of vulnerability trace refinement in enhancing patch-enhanced AST-level signatures, we conducted an experiment comparing PATEN and PATEN-NoVTR, which omits the vulnerability trace refinement step. Table 5 shows the improvement between PATEN and PATEN-noVTR in *PATEN-BENCH*. The inclusion of vulnerability trace refinement in PATEN results in notable improvements across all key metrics. Specifically, precision increases by $6.24\%(=(90.41\%-85.10\%)/85.10\%)$, recall by $5.22\%(=(93.96\%-89.30\%)/89.30\%)$, accuracy by $4.40\%(=(93.87\%-89.90\%)/89.90\%)$ and F1-measure by $5.92\%(=(92.15\%-87.00\%)/87.00\%)$.

PATEN incorporates program slicing to extract relevant variable context subtrees within differential subtrees, focusing specifically on the def-use relationship of variables that contribute to vulnerabilities. For instance, user inputs typically require some form of sanitization before use to prevent malicious injection attacks. To fix such vulnerabilities, additional validation steps are usually inserted between the point of input acquisition and its usage. In this case, the definition-use relationship forms the condition under which

TABLE 5: Performance improvement between PATEN and PATEN-noVTR.

Tool	PATEN-NoVTR	PATEN	Improvement
Precision	85.10%	90.41%	6.24% ↑
Recall	89.30%	93.96%	5.22% ↑
Accuracy	89.90%	93.87%	4.40% ↑
F1-measure	87.00%	92.15%	5.92% ↑

the vulnerability arises, where the definition part occurs before the validation fragment, and the use part follows it. This approach selectively includes only the variable definition prior to the vulnerability patch and the variable usage after it, thereby reducing unnecessary context and minimizing time costs. In contrast, PATEN-NoVTR does not utilize this refinement, potentially leading to a less targeted analysis.

Answer to RQ4: Vulnerability trace refinement in PATEN significantly enhances patch-enhanced AST-level signatures, leading to improved detection of unpatched APIs.

5.6 Discussion

Scalability analysis. In this section, we evaluate the speed and scalability of PATEN in unpatched TPL API discovery. We break down the total time required to discover unpatched TPL APIs into three main components: preprocessing time (the time taken for third-party API extraction), unpatched API detection time (which includes signature generation and matching), and reachability analysis time (the time for API usage detection). Since our task differs from that of the baseline architectures, and we have already discussed the unpatched API detection time for these tools in Section 5.2, we focus here on the specific components of PATEN and their scalability.

We analyzed the detection times for 79 projects that were identified as using unpatched TPL APIs, as shown in Table 3 (including 3 duplicate projects). The lines of code of these projects range from 765 to 789,774. Table 6 presents the time breakdown for unpatched TPL API discovery. The column **AvgT** represents the average time taken for each component, while the column **MaxT** represents the maximum time observed. The preprocessing step takes the longest, with an average time of 94.5 seconds and a maximum of 476.4 seconds, mainly due to the time required for decompiling Java projects. In contrast, the unpatched API detection and reachability analysis steps are significantly faster, with average times of 760.81ms and 105.17ms, respectively. These components contribute minimally to the overall processing time, ensuring scalability and efficiency in large-scale projects.

TABLE 6: Time Breakdown for Unpatched TPL API Discovery

Component	AvgT	MaxT
Preprocessing	94.5s	476.4s
Unpatched API Detection	760.81ms	1,543.32ms
Reachability Analysis	105.17ms	460.81ms

Extensibility. To detect unpatched TPL APIs, we need to match the code snippets of TPL APIs with the corresponding vulnerability-fixing patch to determine whether the API is vulnerable. This paper focuses on the detection of unpatched TPL APIs in Java because Java bytecode, which is platform-independent and can be decompiled back into source code with minimal structural changes, makes it easier to extract unified features for vulnerability detection. This consistency allows for reliable code matching across different platforms. In contrast, C/C++ TPLs are typically compiled into machine-specific binaries, and decompiling them results in significant structural changes. The process is not symmetric, making it much harder to extract consistent TPL source code for vulnerability detection.

Impact of node weight. Inspired by VISION [22], we introduce node weighting to incorporate the frequency differences between nodes in vulnerable and patched API ASTs. The rationale behind this is to emphasize the importance of nodes that undergo significant changes, reflecting their contribution to the vulnerability fix. However, it is important to note that this node weighting mechanism is not the primary contribution of our paper. In our experiments using the *PATEN-BENCH*, the impact of node weighting on performance improvement was minimal, suggesting that the tree edit distance itself, rather than the weighting, plays a more crucial role in detecting unpatched APIs. Nonetheless, we believe that node weighting could be beneficial for future vulnerability detection systems, particularly when dealing with more complex or subtle vulnerabilities where frequency differences may better highlight critical changes.

Contribution and novelty. While existing tools, such as SECURESYNC [14] and VISION [22], also utilize ASTs to represent critical vulnerable code segments, they operate at a coarse-grained level, focusing on entire statement-level subtrees or larger code blocks. No previous studies have explicitly targeted individual node elements within the AST to capture fine-grained vulnerability characteristics. In contrast, our approach introduces a patch-enhanced AST-level signature, which precisely isolates and identifies critical vulnerable elements by extracting and refining node-level changes. By incorporating patch-Induced AST difference extraction and vulnerability trace refinement, we enhance the sensitivity of vulnerability detection, allowing for the identification of nuanced code modifications that directly address vulnerabilities. This fine-grained analysis significantly improves the accuracy of detecting unpatched APIs and better localizes the vulnerability fix.

6 LIMITATIONS OF PATEN

Despite PATEN's advanced capabilities in identifying a significant number of unpatched TPL APIs beyond what is possible with current state-of-the-art tools, there are still instances where it can generate False Negatives and False Positives. We explore these scenarios through two specific examples.

False Negatives Produced by PATEN. While PATEN significantly reduces the number of missed vulnerabilities, about 8% of unpatched TPL APIs still elude detection, especially when APIs contain many similarly structured statements. This limitation, inherent in methods that rely on structural similarity for clone detection, is illustrated in Figure 9.

```
CVE Number:CVE-2015-3192
Bug Fix Commit: 5a711c05ec750f069235597173084c2ee796242
804  private Source processSource(final Source source) {
                                        ......
831      if (xmlReader == null) {
832          xmlReader = XMLReaderFactory.createXMLReader();
833      }
834 +    xmlReader.setFeature("http://apache.org/xml/features/disallow-doctype-
             decl", !isSupportDtd());
835      String name = "http://xml.org/sax/features/external-general-entities";
836      xmlReader.setFeature(name, isProcessExternalEntities());
                                        ......
846  }
```

(a) Patch for fixing CVE-2015-3192

```
Target Version: 4.0.4.RELEASE
Is Vulnerable: True
505  private Source processSource(final Source source) {
                                        ......
529      if (xmlReader == null) {
530          xmlReader = XMLReaderFactory.createXMLReader();
531      }
532      xmlReader.setFeature("http://xml.org/sax/features/external-general-
             entities", this.isProcessExternalEntities());
                                        ......
539  }
```

(b) A target API(in spring-web 4.0.4.RELEASE) that has Vulnerability

Fig. 9: A false negative produced by PATEN.

```
CVE Number: CVE-2020-10719
Bug Fix Commit: aa5e1fe11fec75032f14f0ae23e586f4cf3a3365
65   public void handleRequest(final HttpServerExchange exchange) throws Exception {
66       if (HttpContinue.requiresContinueResponse(exchange)) {
67           exchange.addRequestWrapper(WRAPPER);
68 -         exchange.addResponseCommitListener(NULL)
68 +         exchange.addResponseCommitListener(new ResponseCommitListener() {
69 +             @Override
70 +             public void beforeCommit(HttpServerExchange exchange) {
71 +                 if (!HttpContinue.isContinueResponseSent(exchange)) {
72 +                 exchange.setPersistent(false);
73 +                 IoUtils.safeClose(exchange.getRequestChannel());
74 +                 }
75 +             }
76 +         });
77       }
78       handler.handleRequest(exchange);
79 }
```

(a) Patch for fixing CVE-2020-10719

```
Target Version: 2.2.6
Is Vulnerable: False
62   public void handleRequest(final HttpServerExchange exchange) throws Exception {
63     if (HttpContinue.requiresContinueResponse(exchange)) {
64       exchange.addRequestWrapper(HttpContinueReadHandler.WRAPPER);
65       exchange.addResponseCommitListener(this.createResponseCommitListener());
65     }
66     handler.handleRequest(exchange);
67   }
```

(b) A target API(in undertow-core 2.2.6) that has been patched

Fig. 10: A false positive produced by PATEN.

Figure 9(b) shows the TPL code used in a project named *spring*, which remains unpatched as Line 532 in Figure 9(b) replicates the vulnerable code at Line 836 in Figure 9(a). However, due to the structural similarity with the patched code (Line 834 in Figure 9(a)), it was misclassified as patched, leading to a false negative.

False Positives Produced by PATEN. PATEN also generates some false positives, particularly when projects have replaced a vulnerable TPL with a patched version but have further modified it based on specific requirements. As a result, these revisions might be misinterpreted as indications of vulnerabilities. As shown in Figure 10(a), the official patch applies `addResponseCommitListener` with a direct instantiation of `new ResponseCommitListener()` to define the behavior of the listener explicitly. However, as shown in Figure 10(b), the target code customizes the patch by replacing the direct instantiation with the method call `this.createResponseCommitListener()`, which generates the listener indirectly. However, PATEN does not perform inter-program analysis during preprocessing, and thus cannot resolve the behavior and return value of the method `this.createResponseCommitListener()`. As a result, PATEN incorrectly assumes that the target API is more similar to the vulnerable API, leading to a false positive. Addressing such cases will involve further analysis of the function calls, which is planned for future enhancements of PATEN.

Threats to Validity. The validity of our approach could be compromised by the accuracy and completeness of the *Snyk* platform and other public security sources, which might limit the scope and precision of detected vulnerabilities. Additionally, the limited experimental dataset may not comprehensively represent real-world scenarios, affecting the generalizability and transferability of our results.

Future Work. Moving forward, we intend to enhance the capabilities of PATEN by further refining the AST-level signatures to improve the detection of subtle vulnerabilities. Additionally, we plan to extend our approach to a wider range of programming languages and evaluate its effectiveness across various software ecosystems.

7 Related Work

Code Clone Detection. General techniques for code clone detection, referred to as clone-based approaches in this paper, aim to identify four types of code clones [44], [45], [46]: Type-I and Type-II, which are code snippets that are identical except for variations in spaces, comments, or variable names being renamed [19], [47], [48], [49], [50], [51], [52]; Type-III, which are nearly identical code snippets with a few statements added or deleted [15], [16], [53], [54], [55], [56], [57]; and Type-IV, which are code snippets that perform the same function but have different syntactic structures [17], [58], [59], [60], [61], [62], [63]. While existing techniques for these types are effective in general settings, this paper argues that they often fail to differentiate minor differences between vulnerable code and its patched versions, thus leading to significant false positives when used for vulnerability detection.

Beyond these traditional techniques, several other clone detection methods are designed for vulnerability discovery but do not capitalize on the insights provided by patches. For instance, CP-Miner and its enhanced versions aim to detect bugs by identifying inconsistencies between code clones [64], [65], [66]. SecureSync utilizes extended abstract syntax trees and graph models to represent code snippets, employing these graph representations to identify code reported as vulnerable [14]. Similarly, SCVD employs tree-based matching to detect vulnerable code clones [67], while CBCD detects bugs by generating program dependence graphs and identifying isomorphic sub-graphs [12]. Unlike these techniques, our method fully utilizes patch information, gaining a more detailed and precise understanding of the structural changes caused by patches, thus enabling more effective detection of subtle vulnerabilities that these traditional methods may overlook.

Patch-enhanced Code Clone Detection. To recognize the subtle differences between vulnerable code and its patched versions, previous approaches have attempted to leverage features from both versions [10], [11], [14], [20], [21], [22]. For example, MVP [20] applies program slicing techniques to extract code relevant to vulnerabilities, which are then compared with the target API. MOVERY [21], on the other hand, emphasizes core code lines and function collation to distill key changes during updates, thereby generating concise vulnerability signatures. However, these methods primarily rely on hashing algorithms to encode statement-level characteristics into signatures, but this coarse-grained approach struggles to differentiate between critical and non-critical elements within statements, limiting their accuracy in detecting nuanced vulnerabilities. Unlike these approaches, our method utilizes patch-enhanced AST-level signatures to detect vulnerabilities.

Learning-based Bug Detection. Our approach is related to learning-based bug detection techniques [10], [12], [13], [14], [68], [69] and binary similarity analysis [70], [71], [72], [73], [74], [75], [76]. Learning-based bug detectors typically utilize graph-based representations of code, which are encoded into feature vectors for training machine learning models to classify code as either vulnerable or safe. These models differ from our method, which does not rely on machine learning but rather utilizes direct analysis of code structure and patch information. Conversely, binary similarity analysis operates without access to source code, analyzing binaries directly. This method faces unique challenges due to the lack of rich source code information, making it fundamentally different from our source-level approach. This distinction highlights our method's ability to directly analyze and utilize specific source code features and patch data, providing a distinct advantage over binary-level analysis.

8 Conclusion

In this paper, we have introduced PATEN, a novel approach designed to identify unpatched TPL APIs effectively. By leveraging a fine-grained, patch-enhanced AST-level signature, PATEN significantly surpasses existing state-of-the-art methods in both effectiveness and utility. Our comprehensive evaluations demonstrate the effectiveness and usefulness of PATEN, which has successfully detected 82 critical bugs associated with the use of unpatched TPL APIs in various open-source projects. The source code of PATEN is available at https://github.com/PATEN-Tool/PATEN.

Acknowledgement

We sincerely thank the anonymous reviewers for their valuable and insightful feedback. We also extend our gratitude to Qiao Xiang for his constructive suggestions on the manuscript. This research was supported by the National Key R&D Program of China (2022YFB2901502) and the Natural Science Foundation of China (Grant No. 62272400). Rongxin Wu is the corresponding author and works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University.

References

[1] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *ICSME'18*. IEEE, 2018, pp. 449–460.
[2] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *SANER'15*. IEEE, 2015, pp. 516–519.
[3] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *ESEM'18*, 2018, pp. 1–10.
[4] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *ESE'20*, vol. 25, no. 5, pp. 3175–3215, 2020.
[5] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *ICSME'20*. IEEE, 2020, pp. 35–45.
[6] T. O. W. A. S. Project, "OWASP Dependency-Check," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://owasp.org/www-project-dependency-check/}
[7] S. Inc., "OWASP Dependency-Check," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://www.blackducksoftware.com/}
[8] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *ICSME'18*. IEEE, 2018, pp. 559–563.
[9] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *ICSE-NIER'18*. IEEE, 2018, pp. 101–104.
[10] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 48–62.
[11] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 595–614.
[12] J. Li and M. D. Ernst, "Cbcd: Cloned buggy code detector," in *ICSE'12*. IEEE, 2012, pp. 310–320.
[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.
[14] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *ASE'10*, 2010, pp. 447–456.
[15] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *ICSE'16*, 2016, pp. 1157–1168.
[16] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *ICSE'18*, 2018, pp. 1066–1077.
[17] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE'08*, 2008, pp. 321–330.
[18] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC'08*. IEEE, 2008, pp. 172–181.
[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE'07*. IEEE, 2007, pp. 96–105.
[20] 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 *USENIX Security'20*, 2020, pp. 1165–1182.
[21] S. Woo, H. Hong, E. Choi, and H. Lee, "MOVERY: A precise approach for modified vulnerable code clone discovery from modified Open-Source software components," in *USENIX Security'22*. Boston, MA: USENIX Association, Aug. 2022, pp. 3037–3053. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/woo
[22] S. Wu, R. Wang, K. Huang, Y. Cao, W. Song, Z. Zhou, Y. Huang, B. Chen, and X. Peng, "Vision: Identifying affected library versions for open source software vulnerabilities," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1447–1459.
[23] Snyk, "https://security.snyk.io/vulns," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://security.snyk.io/vulns}
[24] GitHub, "https://github.com/," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://github.com/}

[25] S. Limited, *"https://snyk.io/wp-content/uploads/sooss_report_v2.pdf,"* 2019, [Online; accessed 10-Jan-2024]. [Online]. Available: https://snyk.io/wp-content/uploads/sooss_report_v2.pdf

[26] T. O. W. A. S. Project, "OWASP Top 10," 2013, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf}

[27] ——, "OWASP Top 10," 2017, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://owasp.org/www-project-top-ten/2017/Top_10}

[28] ——, "OWASP Top 10," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://owasp.org/Top10/}

[29] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 679–690.

[30] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On finding lowest common ancestors in trees," *SIAM Journal on Security and Privacy computing*, vol. 5, no. 1, pp. 115–132, 1976.

[31] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[32] V. C. Bhavsar, H. Boley, and L. Yang, "A weighted-tree similarity algorithm for multi-agent systems in e-business environments," *Computational Intelligence*, vol. 20, no. 4, pp. 584–602, 2004.

[33] H. Chim and X. Deng, "A new suffix tree similarity measure for document clustering," in *WWW'07*, 2007, pp. 121–130.

[34] Understand, "https://www.scitools.com/," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://www.scitools.com/}

[35] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE'14*, 2014, pp. 313–324.

[36] Maven, "https://maven.apache.org/," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://maven.apache.org/}

[37] Soot, "https://github.com/soot-oss/soot," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://github.com/soot-oss/soot}

[38] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *FSE'18*, 2018, pp. 319–330.

[39] Maven plugin API, "https://maven.apache.org/ref/3.9.9/maven-plugin-api/," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://maven.apache.org/ref/3.9.9/maven-plugin-api/}

[40] N. I. of Standards and Technology, "National vulnerability database," 2024, retrieved September 10, 2024 from https://nvd.nist.gov/vuln.

[41] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[42] ShiftLeftSecurity, "Joern," https://github.com/ShiftLeftSecurity/joern, 2024, retrieved April 20, 2024.

[43] Clientjava, "https://github.com/prometheus/client_java/issues/705," 2024, [Online; accessed 10-Jan-2024]. [Online]. Available: {https://github.com/prometheus/client_java/issues/705}

[44] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[45] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[46] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *In SCP'09*, vol. 74, no. 7, pp. 470–495, 2009.

[47] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *ICSME'10*. IEEE, 2010, pp. 1–9.

[48] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 387–391.

[49] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[50] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 253–262.

[51] H. Sajnani, V. Saini, and C. Lopes, "A parallel and efficient approach to large scale clone detection," *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 402–429, 2015.

[52] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE'16*, 2016, pp. 87–98.

[53] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSME'1998*. IEEE, 1998, pp. 368–377.

[54] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *ICSE'14*, 2014, pp. 175–186.

[55] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *ICPC'11*. IEEE, 2011, pp. 219–220.

[56] N. Göde and R. Koschke, "Incremental clone detection," in *2009 13th european conference on software maintenance and reengineering*. IEEE, 2009, pp. 219–228.

[57] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 747–769, 2014.

[58] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 81–92.

[59] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *ICSE'11*, 2011, pp. 301–310.

[60] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *SAS'01*. Springer, 2001, pp. 40–56.

[61] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *ICMLA'16*. IEEE, 2016, pp. 1024–1028.

[62] H. Wei and M. Li, "Positive and unlabeled learning for detecting software functional clones with adversarial training." in *IJCAI'18*, 2018, pp. 2840–2846.

[63] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *ICPC'19*. IEEE, 2019, pp. 70–80.

[64] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, "Scalable and systematic detection of buggy inconsistencies in source code," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 175–190.

[65] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *FSE'07*, 2007, pp. 55–64.

[66] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[67] D. Zou, H. Qi, Z. Li, S. Wu, H. Jin, G. Sun, S. Wang, and Y. Zhong, "Scvd: A new semantics-based approach for cloned vulnerable code detection," in *DIMVA'17*. Springer, 2017, pp. 325–344.

[68] 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 *ACSAC'16*, 2016, pp. 201–213.

[69] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[70] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 349–360, 2014.

[71] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *USENIX Security'14*, 2014, pp. 303–317.

[72] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS'16*, vol. 52, 2016, pp. 58–79.

[73] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *CCS'16*, 2016, pp. 480–491.

[74] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on*. IEEE, 2015, pp. 709–724.

[75] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *ACSAC'14*, 2014, pp. 406–415.

[76] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS'17*, 2017, pp. 363–376.

Li Lin is a post-graduate student at the Department of Computer Science and Technology, School of Informatics, Xiamen University. He received his Bachelor's degree in Engineering from Xiamen University in 2022. His current research focuses on software testing and security, with a particular emphasis on addressing the security and reliability issues of databases using software testing techniques, such as metamorphic testing. He has published several papers in top-tier conferences and journals, including ISSTA and TSE. More information about him can be found at: https://matafeiyanll.github.io/

Jialin Ye is a post-graduate student at the Department of Computer Science and Technology, School of Informatics, Xiamen University. He received his Bachelor's degree in Engineering from Shenzhen University in 2023. His current research area is software engineering, with research interests including static analysis and bug detection.

Chao Wang is a Ph.D. candidate at the School of Informatics, Xiamen University, China. His research focuses on software ecosystem management, software security, and software testing, with several publications in top-tier Software Engineering conferences and journals such as ICSE, ASE, and TSE.

Rongxin Wu received the PhD degree from HKUST, in 2017. He is currently an associate professor at the Department of Computer Science and Technology, School of Informatics, Xiamen University. His research interests include program analysis, software security, and mining software repository. His research work has been regularly published in top conferences and journals in the research communities of program languages and software engineering, including POPL, PLDI, ATC, ICSE, FSE, ISSTA, ASE, and TSE and so on. He has served as a reviewer in reputable international journals and a program committee member in several international conferences (FSE'25, ISSTA'25, SANER'25, FSE'24, ISSTA'24, ASE'23, SANER'23, and ASE 2021 and so on). He is a two-time recipient of the ACM SIGSOFT Distinguished Paper Award. More information about him can be found at: https://wurongxin1987.github.io/wurongxin.xmu.edu.cn/