# PPT4J: Patch Presence Test for Java Binaries

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

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

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

David Lo
School of Computing and Information Systems, Singapore Management University
Singapore
davidlo@smu.edu.sg

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

## ABSTRACT

The number of vulnerabilities reported in open source software has increased substantially in recent years. Security patches provide the necessary measures to protect software from attacks and vulnerabilities. In practice, it is difficult to identify whether patches have been integrated into software, especially if we only have binary files. Therefore, the ability to test whether a patch is applied to the target binary, a.k.a. patch presence test, is crucial for practitioners. However, it is challenging to obtain accurate semantic information from patches, which could lead to incorrect results.

In this paper, we propose a new patch presence test framework named PPT4J (**P**atch **P**resence **T**est **for** **J**ava Binaries). PPT4J is designed for open-source Java libraries. It takes Java binaries (i.e. bytecode files) as input, extracts semantic information from patches, and uses feature-based techniques to identify patch lines in the binaries. To evaluate the effectiveness of our proposed approach PPT4J, we construct a dataset with binaries that include 110 vulnerabilities. The results show that PPT4J achieves an F1 score of 98.5% with reasonable efficiency, improving the baseline by 14.2%. Furthermore, we conduct an in-the-wild evaluation of PPT4J on JetBrains IntelliJ IDEA. The results suggest that a third-party library included in the software is not patched for two CVEs, and we have reported this potential security problem to the vendor.

## CCS CONCEPTS

• **Software and its engineering** → *Software reliability*.

## KEYWORDS

Patch Presence Test, Binary Analysis, Software Security

*Corresponding Author

*ICSE '24, April 14–20, 2024, Lisbon, Portugal*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal*, https://doi.org/10.1145/3597503.3639231.

**ACM Reference Format:**
Zhiyuan Pan, Xing Hu, Xin Xia, David Lo, and Xiaohu Yang. 2024. PPT4J: Patch Presence Test for Java Binaries. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639231

## 1 INTRODUCTION

The reuse of open source libraries is widespread [25]. Vulnerabilities in open-source software have become a major concern, posing significant threats to software and users [17]. For example, the number of Common Vulnerabilities and Exposures (CVEs [6]) reported during the first quarter of 2023 has already exceeded the total number of CVEs reported in 2016 [37]. Although upstream developers may discover and fix these vulnerabilities over time, vulnerable versions may still propagate to downstream software or libraries, potentially compromising the security of the systems that rely on them.

Since open source libraries are frequently distributed as binary files, it is essential that developers and users in the software supply chain be aware of potential vulnerabilities in the libraries they introduce. For example, *Log4Shell* (CVE-2021-44228 [33]) is a well-known vulnerability in Apache Log4j [12] that can lead to the execution of arbitrary code. If a software development team has integrated Log4j into their project, developers should verify if their version of Log4j is vulnerable to *Log4Shell*. In other words, they have to confirm whether the binary contains the patch for CVE-2021-44228.

The above process of testing whether a security patch is applied to the program binaries is called *patch presence test* [44]. However, traditional approaches that only take binary files for analysis cannot be utilized for patch presence tests due to the coarse granularity [44]. For example, binary bug search tools, such as Genius proposed by Feng et al. [10], identify vulnerability types but cannot test the presence of an arbitrary patch commit. Similarly, binary code search tools, such as Tracy proposed by David et al. [8], find similar functions but cannot tell whether the function is patched or not.

To accurately test the presence of a patch in fine granularity, Zhang et al. propose FIBER [44], a patch presence test framework for C/C++ binaries that extracts a localized part of patch *diff* [40] for

signature generation. BScout [7], another framework that targets Java binaries, utilizes the entire patch *diff*.

However, existing studies on patch presence test for Java binaries still have the following two limitations:

- **Inability to capture minor changes.** The features extracted by current approaches are unable to handle some subtle modifications to the source code (e.g., changing method call parameters, branch conditions and statements outside method bodies).
- **Limited patch semantic.** The *diffs* utilized in existing approaches cannot perfectly reflect semantic changes (i.e., actual discrepancies in program behavior) in the patches, leading to the inclusion of extraneous information that does not exist in the patches.

Given the widespread application of Java, such as server-side programs and Android applications, the limitations mentioned above highlight the need for a more comprehensive and accurate approach. To address these limitations, we propose a new patch presence test framework named PPT4J (**P**atch **P**resence **T**est **for J**ava Binaries). PPT4J exploits the correspondence between source code features and binary features. It first extracts features from the source code to generate semantic changes. Then, the semantic changes guide PPT4J to perform source-to-binary feature matching and feature queries. Finally, PPT4J provides the test result by summarizing the queries.

To evaluate the effectiveness of PPT4J, we construct a dataset with binaries that include 110 vulnerabilities in total. We compare PPT4J with a state-of-the-art Java patch presence test framework, BScout [7], in terms of accuracy, precision, recall, and F1 score. To evaluate the effectiveness of PPT4J in real-world software, we perform an in-the-wild evaluation by testing the presence of patches in various versions of JetBrains IntelliJ IDEA [38]. The results on the dataset demonstrate that PPT4J achieves an F1 score of 98.5%, improving the baseline by 14.2% while maintaining reasonable efficiency. The results of the in-the-wild evaluation indicate that PPT4J remains accurate in real-world scenarios. In addition, our evaluation reveals that PPT4J has the ability to uncover instances where application vendors (e.g., JetBrains) have failed to apply security patches to third-party libraries they import, further demonstrating its utility in practical settings.

In summary, we make the following contributions:

(1) We propose a novel framework for Java binaries, PPT4J, to accurately test the presence of patches by highlighting semantic code differences from patches.
(2) We construct a dataset to evaluate the effectiveness of PPT4J by obtaining source code from GitHub repositories, labeling ground truths, and building binaries.
(3) We evaluate PPT4J using the dataset and real-world software. PPT4J outperforms the baseline and is able to test the presence of patches in real-world scenarios.
(4) We release the replication package of PPT4J[1], including the source code of PPT4J and our dataset, to facilitate future research.

[1] https://github.com/pan2013e/ppt4j

The remainder of the paper is structured as follows. Section 2 introduces the background concepts for this paper and motivates the problem using an example. Section 3 presents the approach of PPT4J. Section 4 describes the baseline approach, the preparation for our dataset, and the evaluation metrics. Section 5 presents the experimental results and our case study. Section 6 and Section 7 discusses comparsions with the baseline and threats of validity. Section 8 summarizes the related work. Section 9 concludes the paper.

## 2 BACKGROUND

In this section, we discuss the background of the patch presence test task and provide a motivating example to illustrate the main idea of PPT4J.

### 2.1 Patch Presence Test

The task of patch presence test and its scope of the problem are defined as follows:

*Definition.* Given a security patch *diff* $P$ from a specific open source library in the upstream, patch presence test works by evaluating target program binaries $B$ on a boolean function

$$f : (C_1, C_2, P, B) \rightarrow \{\text{True}, \text{False}\},$$

where $C_1$ and $C_2$ refer to the upstream source code of the software right before and after patch $P$. $B$ refer to the standard Java bytecode in this paper and should be provided by the user. Among $C_1, C_2$ and $P$, at least two of them must be provided and the third can be automatically derived.

Patch presence test checks if a specific patch is applied to the provided target binaries [44]. In this task, the upstream source code $C_1, C_2$ and the target binaries $B$ are supposed to belong to the same library, but the target binaries can be compiled from any version of source code. The function $f$ has two possible return values. If $f$ returns true, we confirm the existence of a patch commit within the binaries, and vice versa.

The advantage of patch presence test lies in its ability to detect specific patch commits. This enables users to specifically check for security patches that they are most concerned about. Thus, the potential risks that arise from vulnerabilities can be mitigated.

### 2.2 Motivating Example

Existing patch presence test work for Java binaries takes the complete *diff* (i.e., the exact differences in characters) as input to analyze patch presence. However, existing work is limited as the whole *diff* will introduce "semantic redundancy", that is, some `(-)` and `(+)` lines end up with no semantic changes, which may introduce unrelated information.

Intuitively, to accurately test the presence of a patch, we should extract precise semantic changes from *diff* that reflect all semantic information while not including unrelated information. We illustrate the importance of semantic changes by using real-world examples shown in Figure 1.

In the first case of Figure 1, the *diff* introduces a `try-catch` block with six *diff* lines. However, there are no differences between lines 1, 2, and 5, 6, except for indentations. If we trim the

```
- XmlPullParser        parser = Xml.newPullParser();
- XPPAttributesWrapper attributes = new XPPAttributesWrapper(parser);
+ try
+ {
+   XmlPullParser        parser = Xml.newPullParser();
+   XPPAttributesWrapper attributes = new XPPAttributesWrapper(parser);
```

**(a) CVE-2017-1000498 [28]**

```
- Document<T> doc = parser.parse(is);
+ XMLStreamReader reader = StaxUtils.createXMLStreamReader(is);
+ Document<T> doc = parser.parse(reader);
```

**(b) CVE-2016-8739 [27]**

```
-    if (A == Algorithm.none && B == 2 && C == 0) {
-      return Mapper.deserialize(base64Decode(...), JWT.class);
+    if (B == 2 && C == 0) {
+      if (A == Algorithm.none) {
+        return Mapper.deserialize(base64Decode(...), JWT.class);
+      } else {
+        throw new InvalidJWTSignatureException();
+      }
```

**(c) CVE-2018-11797 [30]**

**Figure 1: Examples of "semantic redundancy": *diff* lines excerpted from full security patches.**

leading spaces/tabs, these lines would be identical. In other words, the actual semantic-related change of the patch is the `try-catch` environment, rather than these four statements. Similar cases include moving statements into `if-else`, `for`, or `while` blocks. In the second case (Figure 1b), lines 1 and 3 differ due to the change of a variable (i.e., `is` ⟶ `reader`). The change originates from calling the method `createXMLStreamReader`. Thus, the code that causes the actual semantic change is line 2, not lines 1 and 3. In the third case (Figure 1c)[2], we can observe that an `if` block is deleted and another `if` block is added. But after inspecting the code, the only semantic change is line 7, that is, throw an exception when `B == 2 && C == 0 && A != Algorithm.none`.

From these examples, we conclude that the differences in characters may not correspond to semantic changes. Motivated by this, we illustrate the idea of our work, which highlights semantic changes and is carried out in the following steps:

Step 1 **Feature extraction.** We extract features for patch-relevant Java source code and binaries.

Step 2 **Semantic change generation.** We generate semantic changes for a patch based on the original *diff* and the extracted features in Step 1.

Step 3 **Feature matching.** We match sequences of source-level features and binary-level features extracted in Step 1.

Step 4 **Patch presence evaluation.** Instructed by semantic changes in Step 2 and feature matching results in Step 3, we evaluated the patch presence for the binaries.

# 3 APPROACH

In this section, we first provide an overview of PPT4J and its architecture. Then, we introduce each component in detail.

[2]The code in the figure is simplified for illustration. The original version is available at https://github.com/FusionAuth/fusionauth-jwt, with commit hash 0d94dcef0133d699f21d217e922564adbb83a227

## 3.1 Overview of PPT4J

The framework of PPT4J is depicted in Figure 2. PPT4J takes the source code and binaries as input. The output of the framework is the patch presence status (i.e., `true` or `false`) of the binaries. Additionally, the framework provides user-configurable parameters, which we will discuss in detail later in this section. By utilizing this framework, PPT4J is able to efficiently analyze binaries and accurately test the presence of patches.

## 3.2 Feature Extraction

This component extracts rule-based features for Java code lines and bytecode blocks.

*3.2.1 Pre-Process.* Raw source code and binaries are not ideal for feature extraction due to certain programming practices, e.g., a statement might be splitted into multiple lines. Additionally, a single instruction may not accurately represent the intended semantic information of a Java statement, which is usually compiled into a group of instructions.

To address this issue, PPT4J incorporates filters that aggregate discrete elements. For instance, we merge split lines in Java source code to create logical lines from abstract syntax trees (ASTs). As for binaries, we propose to split instruction sequences into logical blocks using line number information in binaries. Sometimes, line numbers may be absent or stripped. We discuss such case in Section 6.1.

*3.2.2 Feature Types.* Each line of logical source code or each logical bytecode block corresponds to a set of features that can include zero or more features. Our goal is to select feature types that reflect a large proportion of the Java language specifications and Java VM specifications [1]. To achieve this, we select a variety of simple and non-trivial feature types. This ensures that PPT4J is capable of capturing significant information and minimizing the risk of missing important details. As shown in Figure 3, the feature types in our definition include literals, method invocations, field accesses, array creations, and special instructions.

To be specific, "literal" contains compile-time constant values, fields, and expressions. "Method invocation" contains calls to static methods, virtual methods, and interface methods. We exclude method invocations that can be implicitly generated by the compiler frequently. The excluded methods are `<init>` in `Object` and `StringBuilder`, `toString`, `valueOf`, `append` and `longValue`. For other method invocations, we include a method's name, owner (i.e., the class to which it belongs), and actual parameter types. "Field access" and "array creation", as their names suggest, contain read/write to mutable fields and creation of array objects, respectively. Finally, when extracting "special instructions", we seek source code elements or instructions that meet one of the following characteristics:

(1) Distinctive calculations. ❶ Special binary operators: shift and `instanceof`; ❷ Special unary operators: `++` and `--` (including prefix and postfix).
(2) Control flow manipulations: `return` statements, `throw` statements, `if` blocks and loops.
(3) Synchronization primitives, e.g., `monitorexit` instruction and `synchronized` blocks.

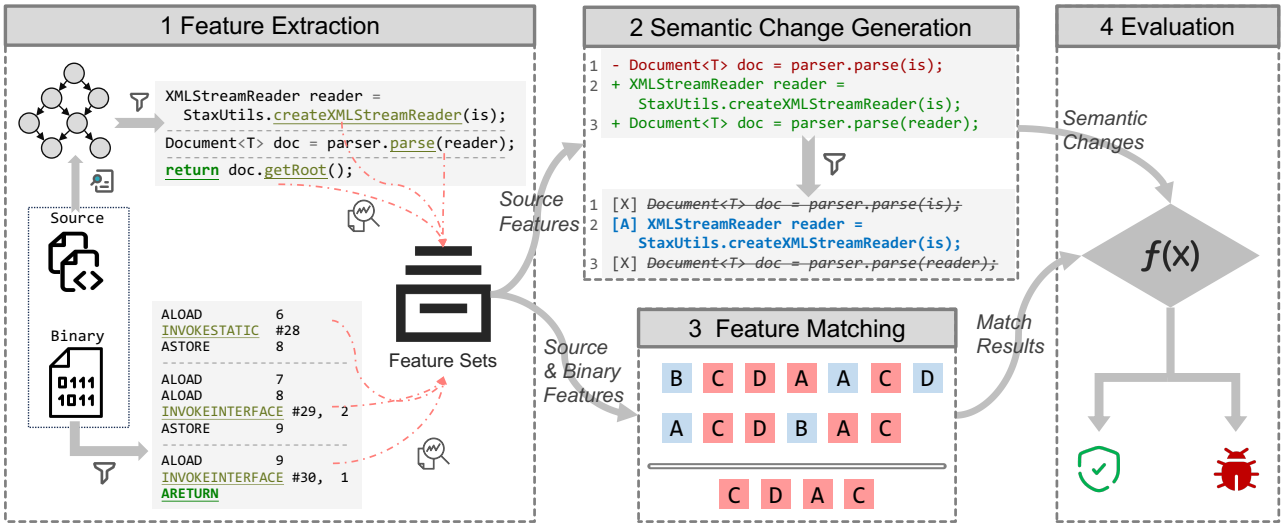


**Figure 2: Overall approach of PPT4J**

(4) Representations of syntatic sugars, e.g., the labels of `switch` blocks, lambda expressions.

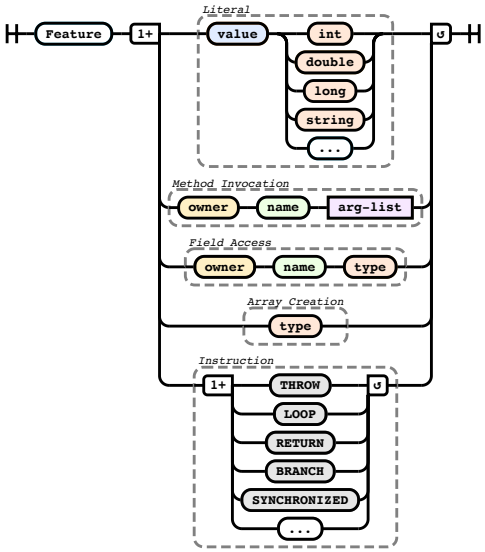


**Figure 3: The railroad diagram that illustrates feature types selected in our approach.**

*3.2.3 Extraction.* For Java source code, we extract features by analyzing ASTs. First, we recursively generate the class dependencies from the `import` statements. Then, we design a custom AST visitor based on Spoon [35]. Specifically, the visitor walks through the syntax tree, from the root node to leaf nodes. If a source code element meets any of the rules described in Section 3.2.2, it extracts the related values or literals, constructs a feature object, and adds it to the corresponding feature set. For Java binaries, we extract features by traversing logical blocks using ASM [3, 20, 23]. Specifically, for each logical bytecode block, we traverse the instructions in the block. Similar to the extraction of source code, if an instruction meets any of the rules, the related values or literals are used to construct a feature object.

For non-trivial feature types, we perform static analysis to collect fine-grained features, especially when dealing with minor patches. The details can be found as follows:

❶ **Literals.** To deal with compiler optimizations on literals, we implement constant propagation and folding for the source code. Accesses to constant fields are replaced by literals, and arithmetics on literals or constant fields are simplified. This transformation is interprocedural and interclass because ASTs of all dependent classes are accessible.

❷ **Method invocations.** Java is an object-oriented programming language, and inheritance is quite common for code reusability. In this case, objects of a derived class can be referenced by variables of a base class. Thus, normal method signatures cannot always reflect the exact types of the arguments of method invocations. To obtain the exact argument types of method calls and build fine-grained method signatures, we implement type analysis for binaries. Starting from the point where an argument variable is initialized, the analysis simulates possible execution paths and returns a more precise type of the variable. To be specific, when walking through a method, it maintains a stack and a local variable table, which store the types of elements[3]. During this process, it takes out variable types in the stack and local variable table, simulates a bytecode instruction, stores the resulting types back to the data structures, and then moves to the next instruction. We use the common superclass if an element type has multiple possibilities. Due to the static type system in Java, it is ensured that the stored type is a subtype of the type found in the method's original signature. Thus, the type analysis results in a more detailed method invocation feature.

[3]If an element is null, we denote the type as null and do not step further.

```
@@ -400,7 +400,7 @@ private static int fastRound(final float x) {
     private int extend(int v, final int t) {
         // "EXTEND", section F.2.2.1, figure F.12, page 105 of T.81
         int vt = (1 << (t - 1));
-        while (v < vt) {
+        if (v < vt) {
             vt = (-1 << t) + 1;
             v += vt;
         }
```

**Figure 4: Patch snippet from CVE-2018-17202 [32]**

**❸ Loop statements.** We distinguish condition statements and loop statements by generating intra-procedure control flow graphs (CFGs) for binaries and analyzing the graphs. For example, the patch in Figure 4 changes a `while` loop to an `if` statement. Although both of `while` loop and `if` statement generate branch instructions, loop statements generate `GOTO` instructions in future blocks that redirect the control flow. Since the Java programming language does not allow programmers to manipulate control flows with keywords like `goto`, we assert that a backward `GOTO` instruction must point to a loop condition.

## 3.3 Semantic Change Generation

As discussed in Section 2.2, commonly used utilities like UNIX *diff* compare texts between different versions. However, *diff* is limited to differences in characters and cannot explain the behavior of the program [15]. To evaluate patch presence more accurately, we extract semantic changes for each security patch based on the original *diff* generated from the `git diff <commit> <commit>` command. This command compares two commits of a git repository and generates output in *unified diff format* [18].

First, we parse the *diff* and split the hunks (i.e., groups of differing lines interspersed in files) into finer-grained code blocks. During the parsing process, we filter out unrelated *diff* contents, i.e., lines that contain no features. Also, split lines are treated as a single logical line. After that, we categorize the blocks into three types, *pure addition* (***A***-type), *pure deletion* (***D***-type), and *mixed blocks* (***M***-type). An ***M***-type block consists of both `(+)` lines and `(-)` lines, in which two kinds of lines overlap or adjoin in space, and `(-)` lines appear before `(+)` lines.

Then, we classify the *diff* lines as pure `(+)` lines, pure `(-)` lines and {`(-)` line, `(+)` line} pairs, and put them in different sets. The classification result reflects the semantic of the patch and can be used to instruct the patch presence evaluator (in Section 3.5). We define four types of sets: **SA** for pure `(+)` lines, **SD** for pure `(-)` lines, **SX** for lines to be excluded in our approach, and **SM** for {`(-)` line, `(+)` line} pairs. In particular, **SX** refers to the lines that do not participate in future evaluations. The line pairs in the **SM** set are called "modification line pairs". We consider all *diff* lines as semantic changes for ***A***-type and ***D***-type blocks and put them into **SA** or **SD** sets. We apply a heuristic approach for ***M***-type blocks using previously extracted features to filter semantic changes, as described in Algorithm 1. The above four sets contain semantic change lines and are utilized in the patch presence evaluation (Step 4 in Figure 2).

**Algorithm 1** Filtering semantic changes for ***M***-type blocks

**Input:** An ***M***-type diff block B

```
1:  del, add ← splitByType(B)
2:  c1 ← getSmallerOne(del, add)
3:  c2 ← getLargerOne(del, add)
4:  window_size ← c1.size()
5:  initWindow(c2, window_size) ▷ Initialize a sliding window on c2
6:  for all possible windows do
7:      best_window ← the most similar[4] window to c1.
8:      coeff ← simMetric(c1, best_window)
9:  end for
10: if coeff = 1 then                    ▷ These lines should be excluded
11:     SX.putAll(c1)
12:     SX.putAll(best_window)
13: else if coeff ≥ σ_f then             ▷ Considered as modification pairs
14:     makePairs(c1, best_window).forEach(SM::put)
15: end if
16: for all remaining line l do
17:     if l.type = (+) then
18:         SA.put(l)                    ▷ Considered as addition lines
19:     else
20:         SD.put(l)                    ▷ Considered as deletion lines
21:     end if
22: end for
```

To be specific, Step 2 in Figure 2 illustrates an example of semantic change generation. In this example, we can observe that lines 1 and 3 do not have semantic changes if we put them together, and thus should be excluded. Line 2, on the other hand, has a semantic change (i.e., function call `createXMLStreamReader`), thus we put it into the **SA** set.

Our semantic change generator can handle all code differences inside method bodies. However, for out-of-method source code lines, we only consider assignments to fields. Since these assignments are components of the constructor methods `<init>` or static initialization blocks `<clinit>`, which cannot be ignored. In this paper, we ignore other types of out-of-method code differences, such as modifying method signatures or implementing new interfaces and we assume that these changes should be reflected by other changes inside the method bodies.

## 3.4 Feature Matching

From the last step, we extract unified features from the source code and the binaries, they can be considered as sorted sequences of feature sets, i.e.,

$$[(idx_1, S_1), (idx_2, S_2), (idx_3, S_3) \cdots]$$

where $idx_1 < idx_2 < \cdots < idx_n$.

$idx_i$ refers to a logical line number in the source feature sequence or an instruction block index in the binary feature sequence, and $S_i$ refers to a feature set. For source sequences, $idx$ may not be consecutive due to empty or comment lines. In practice, we remap these indices for convenience.

Inspired by sequence alignment techniques used in other disciplines, we employ a similar approach to match source code features

[4] Similarity metric for feature sets and threshold parameter $\sigma_f$ are discussed in Section 3.4

and binary features. In general, given two feature set sequences,

$$A = [(a_1, S_{a_1}), \cdots, (a_n, S_{a_n})],$$
$$B = [(b_1, S_{b_1}), \cdots, (b_n, S_{b_n})],$$

the feature matcher outputs a sequence

$$C = [(a_{i_1}, b_{j_1}, S_{k_1}), \cdots, (a_{i_n}, b_{j_n}, S_{k_n})]$$

where the first key in the tuple sorts elements. The $a_{i_n}$-th logical source code line matches the $b_{j_n}$-th binary block. Before matching, we first define the equivalence of two feature sets:

*Definition.* Given two finite feature sets $S_1, S_2$ and a similarity metric $f : (S_1, S_2) \rightarrow [0, 1]$, $S_1$ and $S_2$ are equivalent iff

$$f(S_1, S_2) \geq \sigma_f, \sigma_f \in (0, 1)$$

where $\sigma_f$ is a threshold parameter. By default, $\sigma_f$ is set to 0.7.

Specifically, we use the Jaccard similarity coefficient [41] $\mathcal{J}$ as the similarity metric, that is

$$\mathcal{J}(A, B) = \begin{cases} 1 & \text{A and B are both empty} \\ \dfrac{|A \cap B|}{|A \cup B|} & \text{otherwise} \end{cases}$$

After defining the equivalence of feature sets, we apply a sequence-matching algorithm. Specifically, we use the longest common subsequence (LCS) algorithm because we should obtain as many matches as possible and we expect the matches to be in order. For example, Step 3 in Figure 2 illustrates the matching process of two sequences `BCDAACD` and `ACDBAC`. After applying the LCS algorithm, we get the matching result `CDAC`. In our LCS algorithm, the elements in the sequences are feature sets. Then, the aforementioned similarity metric and equivalence of two feature sets are applied.

However, due to line breaks in the source code, some source feature sets in the sequence are not matched with binary feature sets. To address this issue, after running the LCS algorithm, we start a second matching pass for unmatched binary blocks and employ a heuristic approach that attempts to search for the best match. The heuristic idea is that the union of a few consecutive unmatched binary feature sets is likely to match a source feature set. Specifically, we employ a variable-length sliding window to scan the spaces of the unmatched binary feature sets. As the sliding window moves, in each move we obtain an *aggregated feature set*, which simply means the union of all binary feature sets in a window. If an aggregated feature set corresponds to a source code line, we regard it as a new match. Since the scanning process is time-consuming and memory-consuming and we assume that programmers generally avoid splitting statements into too many lines, we limit the maximum window size to increase performance. By default, the size is set to 5 instructions/window.

## 3.5 Patch Presence Evaluation

The patch presence evaluator serves as the decision-making component in our approach, responsible for aggregating all relevant information and determining the presence of a patch in the binaries. The idea is to assign votes to each semantic change line and let them make recommendations. These semantic change lines are obtained from the output of the semantic change generator as described in Section 3.3. Our voting rules for the patch presence evaluator are described below:

❶ There are two voting options: **A** for those in favor of the binary being **patched**, and **B** for those in favor of the binary being **unpatched**.

❷ For the `(+)` lines in the **SA** set, if the lines appear in the binaries, they vote for **A**. If not, they vote for **B**.

❸ Similarly, for the `(-)` lines in the **SD** set, if the lines **do not** appear in the binaries, they vote for **A**. Otherwise, they vote for **B**.

❹ For modification line pairs `(pre, post)` in the **SM** set, we expect the features in binaries to be more similar to the post-patch line than to the pre-patch line. Thus, if the similarity score of (`post`, binary) is larger than (`pre`, binary), the lines vote for **A**. Otherwise, they vote for **B**. In terms of similarity, we use the same metric described in Section 3.4.

❺ Lines in the **SX** set are ignored as they do not contain semantic changes.

To determine whether a given source code line appears in the binaries, we check the existence of its corresponding logical line number in the output sequence generated by the feature matcher. In addition to these rules, lines with a greater number of features are preferred in our approach. To be specific, the number of votes for each line is equivalent to the number of features it possesses. Once we have all the votes, we calculate the patch's "support rate", which refers to the percentage of total votes in favor of the binary being patched. We consider the patch to be present if the support rate equals or exceeds a specified threshold parameter, denoted by $\sigma_p$. By default, the parameter is set to 0.6.

# 4 EXPERIMENTAL SETTINGS

In this section, we describe the details of the baseline, dataset, and implementation of our approach.

## 4.1 Baseline

There exist a number of patch presence test approaches [7, 21, 39, 44]. However, most of them target C/C++ binaries and cannot be directly employed to test Java binaries because machine instruction sets and bytecode instruction sets differ a lot. Among the existing approaches, BScout [7] is a patch presence test framework specifically designed for Java binaries, and we use it as our baseline. Since the official implementation of BScout is not publicly available, we reimplement their approach with about 5,000 lines of Java code.

## 4.2 Dataset

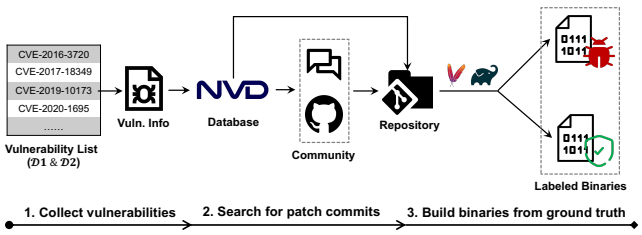


**Figure 5: Steps to construct the dataset**

To evaluate the effectiveness of Ppt4J, we construct a dataset, as illustrated in Figure 5. In detail, the construction process can be divided into the following three steps:

**Table 1: List of vulnerabilities selected from Vul4J [4]**

| Library Name | Vulnerability ID | Library Name | Vulnerability ID |
|---|---|---|---|
| alibaba/fastjson | CVE-2017-18349 | eclipse/rdf4j | CVE-2018-20227 |
| apache/camel | CVE-2015-0264, CVE-2015-0263 | ESAPI/esapi-java-legacy | CVE-2013-5960 |
| apache/commons-compress | CVE-2019-12402, APACHE-COMMONS-001 | esigate/esigate | CVE-2018-1000854 |
| apache/commons-configuration | CVE-2020-1953 | FasterXML/jackson-dataformat-xml | CVE-2016-7051, CVE-2016-3720 |
| apache/commons-fileupload | CVE-2013-2186 | inversoft/prime-jwt | CVE-2018-1000125, CVE-2018-1000531 |
| apache/commons-imaging | CVE-2018-17201, CVE-2018-17202 | javamelody/javamelody | CVE-2013-4378 |
| apache/commons-io | CVE-2021-29425 | jenkinsci/ccm-plugin | CVE-2018-1000054 |
| apache/cxf | CVE-2015-5253, CVE-2016-8739 | jenkinsci/groovy-sandbox | CVE-2018-1000865 |
| apache/httpcomponents-client | HTTPCLIENT-1803 | jenkinsci/jenkins | CVE-2017-1000355, CVE-2018-1000864, CVE-2018-1999044 |
| apache/jspwiki | CVE-2019-0225 | jenkinsci/junit-plugin | CVE-2018-1000056 |
| apache/pdfbox | PDFBOX-3341, CVE-2018-11797 | jenkinsci/pipeline-build-step-plugin | CVE-2018-1000089 |
| apache/santuario-java | CVE-2014-8152 | jenkinsci/subversion-plugin | CVE-2018-1000111 |
| apache/shiro | CVE-2016-6802 | neo4j-contrib/neo4j-apoc-procedures | CVE-2018-1000820 |
| apache/sling | CVE-2016-5394, CVE-2016-6798 | OpenRefine/OpenRefine | CVE-2018-20157, CVE-2018-19859 |
| apache/sling-org-apache-sling-xss | CVE-2017-15717 | resteasy/Resteasy | CVE-2020-1695 |
| apache/struts | CVE-2016-8738, CVE-2014-0113, CVE-2014-0112, CVE-2016-4465, CVE-2016-2162, CVE-2014-7809, CVE-2016-4436, CVE-2014-0116, CVE-2015-1831, CVE-2016-3081 | spring-projects/spring-data-rest | CVE-2017-8046 |
| apache/tika | CVE-2018-8017 | spring-projects/spring-framework | CVE-2018-15756, CVE-2016-9878, CVE-2018-1272 |
| apache/tomee | CVE-2015-8581 | spring-projects/spring-security-oauth | CVE-2016-4977 |
| apereo/java-cas-client | CVE-2014-4172 | square/retrofit | CVE-2018-1000850 |
| cloudfoundry/uaa | CVE-2019-3775, CVE-2018-1192 | swagger-api/swagger-parser | CVE-2017-1000207 |
| codehaus-plexus/plexus-archiver | CVE-2018-1002200 | x-stream/xstream | CVE-2019-10173 |
| (Continued in the right column) | | zeroturnaround/zt-zip | CVE-2018-1002201 |

**❶ Vulnerability collection.** In this step, we collect a number of vulnerabilities for evaluation. We include two lists of vulnerabilities. The first list (abbreviated as $\mathcal{D}\mathbf{1}$) consists of the Java library vulnerabilities evaluated in BScout's experiments, and the second one (abbreviated as $\mathcal{D}\mathbf{2}$) is selected[5] from Bui et al.'s work, i.e., Vul4J [4]. The corresponding patches in $\mathcal{D}\mathbf{1}$ are trivial and are used to check the correctness of our reimplementation of BScout. To evaluate the ability to test the presence of patches that introduce minor changes, we also include the $\mathcal{D}\mathbf{2}$ list. Detailed vulnerabilities in $\mathcal{D}\mathbf{2}$ are listed in Table 1.

**❷ Commit searching.** In addition to the list of vulnerabilities, the dataset also contains project snapshots and corresponding binaries labeled with ground truth. For a vulnerability in the list, we first search it in the National Vulnerability Database (NVD) [26], a CVE database managed by the U.S. government. If a patch commit URL is given in the database, we are done with the search. Otherwise, we manually search for patch-related commit IDs or issue IDs in open-source projects. Once we locate the URL, we can derive the patch status (i.e., patched or unpatched) of a project snapshot after searching. We use patch status as ground truth to label a project snapshot.

**❸ Binary compilation.** In the final step, we compile binaries from project snapshots. The compilation process can be automated as most open-source Java projects are well documented and provide build scripts , e.g., `build.xml` [11], `pom.xml` [13], `build.gradle` [19] and shell scripts. In practice, most of the projects in the dataset [4] can be built without human intervention, except for a few old versions of the Spring Framework, since some dependencies cannot be automatically resolved due to the change of their repository URLs[6]. To correspond to the common compilation and distribution process for open source libraries, we use the default compiler options and flags specified in the build scripts.

[5]Some vulnerabilities are excluded because the corresponding projects are too old and the build tools fail to resolve some dependencies during compilation.

[6]https://spring.io/blog/2020/10/29/notice-of-permissions-changes-to-repo-spring-io-fall-and-winter-2020

## 4.3 Evaluation Metrics

Patch presence test can be considered as a binary classification problem in evaluations, and there are four possible outcomes:

- True Positive (**TP**): The binaries are patched, and PPT4J detects the presence of the patch.
- True Negative (**TN**): The binaries are not patched, and PPT4J does not detect the presence of the patch.
- False Positive (**FP**): The binaries are not patched, but PPT4J detects the presence of the patch.
- False Negative (**FN**): The binaries are patched, but PPT4J does not detect the presence of the patch.

Based on these possible outcomes, the evaluation metrics we use are defined as follows:

**Accuracy** is the proportion of correct predictions among the total number of cases examined.

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

**Precision** measures the accuracy in classifying a sample as positive.

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

**Recall**, also known as True Positive Rate (**TPR**), measures the ability to detect positive samples.

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

**F1 Score** is the harmonic mean of precision and recall, which symmetrically represents both precision and recall in one metric.

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

## 4.4 Implementation of Our Approach

We implement the framework in Java and the replication package is publicly available.[7] To parse Java source code and analyze ASTs,

[7]https://github.com/pan2013e/ppt4j

we exploit Spoon which is a library for implementing analyses and transformations of Java source code proposed by Pawlak et al. [35]. To parse and analyze Java binaries, we employ ASM [20], which is an all-purpose Java bytecode manipulation and analysis framework. We set the threshold of feature equivalence $\sigma_f$ to 0.7 and the threshold of patch support rate $\sigma_p$ to 0.6. Besides, we set the maximum window size of the second feature matching pass to 5 instructions. PPT4J is lightweight and does not cost too much CPU and memory resources. We perform our experiments on a Java HotSpot(TM) 64-bit server VM (build 17.0.2+8-LTS-86) on a personal computer (MacOS 13, 64-bit 3.2GHz CPU, 16GB RAM).

## 5 RESULTS

In this paper, we aim to answer the following four research questions:

**RQ.1 (Effectiveness)** How accurate is the patch presence test framework compared to previous work?

**RQ.2 (Efficiency)** How efficient is the patch presence test framework, especially when dealing with large code repositories?

**RQ.3 (Ablation Study)** How do the analyses described in Section 3.2.3 contribute to the overall effectiveness?

**RQ.4 (In-the-wild Evaluation)** Can our approach analyze open-source libraries in real-world applications?

### 5.1 RQ1: Effectiveness

**Table 2: Test results on the dataset**

| Test Suite | | Metrics | | | |
|---|---|---|---|---|---|
| | | Acc. | Prec. | Recall | F1 |
| BScout | $\mathcal{D}1$ | 100% | 100% | 100% | 100% |
| | $\mathcal{D}2$ | 87.9% | 100% | 75.8% | 86.2% |
| PPT4J | $\mathcal{D}1$ | 100% | 100% | 100% | 100% |
| | $\mathcal{D}2$ | 98.5% | 100% | 97.0% | 98.5% |

*5.1.1 Results.* The overall result is presented in Table 2. According to Table 2, we can observe that both BScout and our work achieve 100% accuracy, precision, recall and F1 score on $\mathcal{D}1$. The results of BScout illustrate that our reproduction of it is consistent with the results reported by Dai et al [7]. When we perform experiments on $\mathcal{D}2$, which has more subtle patches, the recall and F1 score of BScout drop noticeably, while PPT4J still maintains a remarkable performance.

In addition to the conclusion mentioned above, PPT4J does not generate false positive results, which means that it does not mistakenly report a binary as having been patched when it has not. This is an important feature as it ensures that developers can trust the tool's output and avoid wasting time investigating false leads.

*5.1.2 Qualitative Analysis.* We also conduct an qualitative analysis that presents some representative patterns to demonstrate the strengths of our work compared to BScout.

❶ **Minor changes.** We examine several security patches in our dataset and find that the features extracted by BScout are unable to distinguish some minor changes.

```
@@ -174,7 +174,7 @@ public <T> T deserialze(DefaultJSONParser parser, Type
↪  type, Object fieldName) {
            componentType = componentClass = clazz.getComponentType();
        }
        JSONArray array = new JSONArray();
-       parser.parseArray(componentClass, array, fieldName);
+       parser.parseArray(componentType, array, fieldName);

        return (T) toObjectArray(parser, componentClass, array);
    }
```

**Figure 6: Patch snippet from CVE-2017-18349 [29]: An example of minor change that only replaces an argument**

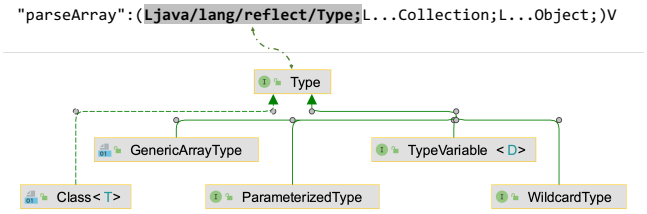


**Figure 7: Class hierarchy of `java.lang.reflect.Type`**

For example, Figure 6 shows the patch *diff* of CVE-2017-18349. In this patch, only the first parameter of the method call `parseArray(Type, Collection, Object)` is changed from `componentClass` to `componentType`. This is a minor change that only includes a variable replacement.

👎 **BScout:** According to Dai et al. [7], BScout only extracts method names with argument lengths. In this case, the features of the method call before and after the patch are identical.

👍 **PPT4J:** In this case, PPT4J works because it implements a type analysis, as described in Section 3.2.3. The type of the first parameter in the method signature is `java.lang.reflect.Type`. It is too general in the class hierarchy, and many subclasses implement this interface, as illustrated in Figure 7. With the help of type analysis, PPT4J traces the variables and checks their exact type instead of using their common super class `java.lang.reflect.Type`. The exact type of `componentClass` is `java.lang.Class`, and the exact type `componentType` is `java.lang.reflect.Type`. In this way, PPT4J extracts different features before and after the patch, and thus detects the minor change.

❷ **Syntactic sugars.** Syntatic sugars [42] make modern programming languages carry more semantic information than their lower-level representations such as IR, bytecode, and assembly code. To accurately perform feature matching (as described in Section 3.4), we must bridge the gaps between source code and binary, i.e., interpret syntactic sugars in the source code to bytecode representations. PPT4J is able to interpret and capture these types of information to reduce the number of false negative results.

For example, variable-length arguments (abbriviated as *varargs*) are syntactic sugars in Java and are converted into raw arrays in code generation. BScout avoids extracting array creations because Java compilers automatically create `java.lang.Object[]` when calling *vararg* methods [7]. This solves the inconsistency problem, but BScout cannot detect if a programmer manually creates an array. On the contrary, PPT4J interprets *varargs* at the source code level, allowing it to detect all kinds of array creations.

```
@@ -46,7 +47,7 @@ public abstract class MimeTypeUtils {
    // other codes omitted

-   private static final Random RND = new Random();
+   private static final Random RND = new SecureRandom();
```

**Figure 8: Patch snippet from CVE-2018-1272 [31]: An example of syntatic sugar that uses a field initialization statement**

**Table 3: Time consumption on the dataset**

| Framework | Average | ~75%[a] |
|---|---|---|
| BScout[b] | 0.34 sec/patch | 0.28 sec/patch |
| PPT4J | 0.48 sec/patch | 0.30 sec/patch |

[a] 75% of test cases are analyzed within this amount of time.
[b] This refers to our reproduction of Dai et al.'s work [7].

Another example of syntactic sugars in Java is field initializations. Field initializations are actually statements in the constructor method `<init>` or static initializing block `<clinit>`. Our work PPT4J is designed to extract features from field initializations, but BScout simply ignores them. Figure 8 is a security patch of the Spring framework [34], which only changes the initialization of the field `RND`. PPT4J detects this change, while BScout does not.

❸ **Semantic redundancy in *diff*.** As shown in Section 2.2, PPT4J utilizes the features of the source code and excludes unrelated information in *diff*. However, BScout takes into account the whole patch *diff*. Although BScout indeed does not miss any semantic information in the patches, it is very likely to introduce unrelated information and fail in cases as shown in Figure 1c.

> **Answer to RQ1**: PPT4J achieves high accuracy, precision, recall and F1 score on the dataset (98.5%, 100%, 97% and 98.5% respectively) and outperforms our baseline BScout by 14.2% in terms of F1 score. In addition, PPT4J is effective in handling patches with minor changes.

## 5.2 RQ2: Efficiency

To answer RQ2, we measure the time consumption of PPT4J on the dataset.[8] Table 3 shows that most patches can be quickly analyzed. Some patches may take a bit longer (several seconds) because of the CFG construction and analysis on large bytecodes, but we think it is still acceptable compared to the time costs of human inspections. PPT4J can achieve this performance because only dependent bytecodes are analyzed, so the time cost is not proportional to the project size. Besides, the preprocessing phase in dataset preparations caches the source code features and reduces the analysis time when users input their binaries. Table 3 also lists the time consumption of our reimplemented baseline, BScout. Compared to BScout, PPT4J is a bit slower in most test cases, but we think that PPT4J's advantages in effectiveness (as discussed in RQ1) can compensate for the shortcomings in terms of time efficiency.

[8]When collecting data, we ignore the startup time of the Java virtual machine and third-party dependencies, and only focus on the components described in Section 3.

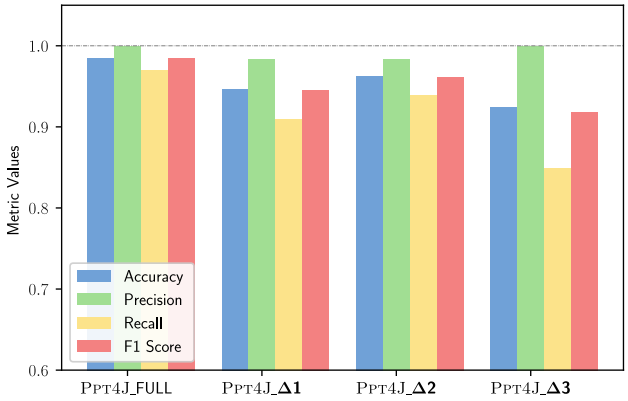


**Figure 9: Test results for different variants of PPT4J**

> **Answer to RQ2**: PPT4J analyzes most security patches in one second and is also fast when analyzing binaries from large projects.

## 5.3 RQ3: Ablation Study

To investigate the contribution of an analysis and its corresponding feature type (as described in Section 3.2.3) to the effectiveness of PPT4J, we create the following variants:

- PPT4J_FULL: The complete version of PPT4J, exactly as Section 3 illustrated.
- PPT4J_**Δ1**: Type analysis is removed. This means that method signatures are not revised during feature extraction.
- PPT4J_**Δ2**: Some special instructions are ignored, such as loop and branch.
- PPT4J_**Δ3**: Constant propagation/folding is removed. This means that constant fields and expressions are not simplified during feature extraction.

We evaluate the above four variants on the same dataset (as described in Section 4.2), and compare the incomplete ones with PPT4J_FULL. The results are illustrated in Figure 9. We can conclude that PPT4J benefits from these analyses. In the cases of PPT4J_**Δ1-3**, the F1 score decreases by 4.0%, 2.3%, and 6.8% respectively after one specific analysis is removed.

> **Answer to RQ3**: The analyses proposed in Section 3.2.3 make extracted features fine-grained, and they indeed improve the effectiveness of PPT4J.

## 5.4 RQ4: In-the-wild Evaluation

*5.4.1 Experimental setting.* To evaluate the effectiveness of our approach in practical settings, we use IntelliJ IDEA Ultimate [38] as the target software for our in-the-wild evaluation. The reason to choose IntelliJ IDEA is that it is a widely used IDE software, and it embeds a large number of binaries from open-source Java libraries. We obtain various versions of IntelliJ IDEA and extract the included third-party libraries. Next, we conduct patch presence

**Table 4: Evaluation on different versions of IntelliJ IDEA**

| Vulnerability | Version Timeline[a] | | | | |
|---|---|---|---|---|---|
| | V1 | V2 | V3 | V4 | V5 |
| CVE-2019-12402 :08/19[b] | TN | TN | TP | TP | TP |
| CVE-Anonymous-1 | TN | TN | TN | TN | TN |
| CVE-Anonymous-2[c] | TN | TN | TN | TN | TN |
| CVE-2021-29425 :05/18 | TN | TN | TP | TP | TP |
| HTTPCLIENT-1803 :01/17 | TN | FN | FN | FN | FN |
| CVE-2017-1000487 :10/13 | TP | TP | TP | TP | TP |
| CVE-2015-6748 :07/15 | N/A[d] | TP | TP | TP | TP |
| CVE-2015-6420 :11/15 | TN | TP | TP | TP | TP |

[a] V1 - V5 are 5 versions of IntelliJ IDEA Ultimate, sorted in ascending order of release time. V1: IU-181.5684.4; V2: IU-191.8026.42; V3: IU-203.8084.24; V4: IU-213.7172.25; V5: IU-231.8109.175. The first two digits in the version string specify the release year, e.g., V1 was released in 2018.
[b] Patch commit time. Retrieved from Github, in MM/YY format.
[c] Real IDs of *CVE-Anonymous-1/2* are omitted due to "responsible reporting" principle.
[d] N/A means this version of software does not include the library.

tests on these open-source libraries for specific patches. However, labeling real-world binaries with ground truth is challenging since not all binary packages contain version information. To address this issue, we utilize unit tests in some patches. These unit tests are committed together with the bug fixes, and can be trusted sources of ground truths. If our patch presence predictions align with the unit test results, it indicates that our approach can perform well in real-world scenarios. Specifically, each test case employed in this experiment includes: ❶ Binaries of a third-party library extracted from IntelliJ IDEA. ❷ A security patch, along with the unit test, and the source code of the library before and after the patch.

*5.4.2 Results.* The detailed result is shown in Table 4. We conclude that most of the output from Ppt4J is consistent with the unit tests and the accuracy is 89.7%. We also notice that Ppt4J does not generate false positive outputs. In conclusion, we believe that Ppt4J is capable of real world scenarios. We also learn some facts about the patch status in IntelliJ IDEA from the experiment results.

❶ The application vendor JetBrains promptly applied the upstream patch for CVE-2019-12402, ensuring that libraries in releases after 2020 were not affected by this vulnerability. However, even though the patch commit for CVE-2021-29425 was released in May 2018, JetBrains did not apply it in one of their 2019 versions, such as *IU-191.8026.42*. Similarly, for CVE-2015-6420, the patch commit was available as early as November 2015, but they failed to apply it in one of the 2018 versions.

❷ We notice that a third-party library in IntelliJ IDEA has not yet been patched for two specific vulnerabilities (i.e., *CVE-Anonymous-1* and *CVE-Anonymous-2*, as shown in Table 4) until now. The application vendor JetBrains forked its branch, but has not merged the upstream branch later, leaving the vulnerabilities unresolved. Although we cannot definitively assert that vulnerabilities in open source libraries will invariably impact commercial software, we believe that minimizing vulnerabilities in these libraries can mitigate the risk of exploitation. Thus, we report this problem to JetBrains.

❸ We also perform this evaluation using our reimplemented baseline. Compared to Ppt4J's results, the accuracy of BScout decreases by 14.3% and drops to 76.9%. Upon examination of the failed test cases, we find out Ppt4J performs better due to its effectiveness in handling patches with minor changes, as discussed in RQ1. For example, in the case of *CVE-Anonymous-1*, BScout cannot even tell whether a binary contains the patch because no features are extracted from the patch.

**Answer to RQ4**: Ppt4J is capable of analyzing open source libraries in real-world applications. In our evaluation, it achieves an accuracy of 89.7% and detects two unpatched CVEs in a third-party library within IntelliJ IDEA. We have reported this potential problem to JetBrains.

# 6 DISCUSSION

## 6.1 Line number information

Although generated by default, line number information is part of the debug information, and can be stripped from binaries. Ppt4J fails without such information. Despite this, we perform an empirical study on line number information in open-source Java libraries. We collect 9,077 jar files (containing 2,032,221 class files in total), which are downloaded from the Maven central repository [13]. Our findings show that over 90% of these class files include line number information. As Ppt4J is proposed for use for general open-source libraries, we believe it is still applicable in many practical scenarios.

## 6.2 Ppt4J vs. Baseline

Compared to the baseline BScout, Ppt4J selects different sets of feature types, and applies different algorithms to extract non-trivial features and handle syntactic sugars. With these algorithms, Ppt4J is able to capture minor changes in patches and the extracted features are more fine-grained, as illustrated in Section 3.2.3 and the qualitative analysis on the dataset in Section 5.1. It is also worth noting that we propose the semantic change generator in Ppt4J. With semantic change generation, Ppt4J is able to process *diff* files and filter out unrelated information based on the previously extracted features, as illustrated in Section 2 and Section 3. However, BScout takes the complete patch *diff* as input and cannot handle the semantic redundancy in diffs. The drawback of Ppt4J compared to BScout is its efficiency and dependence on line number information, but as discussed earlier in Section 5.2 and Section 6.1, we believe Ppt4J remains practical.

# 7 THREATS TO VALIDITY

**Correctness of our reproduction.** ❶ As mentioned earlier in Section 4.2 and Section 5.1, we reimplement BScout [7] and reproduce identical results on dataset $\mathcal{D}1$ in terms of accuracy, precision, recall and F1 score. However, in addition to $\mathcal{D}1$, Dai et al. [7] also construct a dataset consisting of Android applications and evaluate their approach on this dataset. Since the details of this dataset are not available, we cannot reproduce all experiments in the authors'

paper. This might be a threat to the correctness of our reproduction and the effectiveness evaluation. ❷ We only reimplement one variant of BScout that relies on line number information. The complete version of BScout leverages a machine learning model, but neither the model weights nor the training dataset is publicly available. Thus, it is not practical for us to reimplement the complete one. However, this does not create issues with the reliability of our findings in Section 5. According to Dai et al. [7], the performance of the complete version of BScout is lower than the variant if line number information is present. Therefore, the performance of our reimplemented baseline reported in Section 5 is supposed to be an upper bound.

**Patch backporting.** Backporting security patches to older versions is a common practice, which means that multiple different patch commits can correspond to the same vulnerability. However, when constructing the dataset, we only selected one version for each vulnerability, and Ppt4J cannot automatically select the correct patch commit. This limitation can lead to erroneous results when Ppt4J analyzes open-source libraries in real-world scenarios.

**Inherent flaws with rule-based features.** Ppt4J extracts rule-based features from the source code, providing finer granularity than the baseline approach. However, this method has an inherent limitation in that it cannot always cover all language features. As the Java language continues to evolve, the tool must be updated to adapt to programs written using new standards.

**Effectiveness against version changes.** In Ppt4J, we use the longest common subsequence algorithm to match the features. However, this algorithm assumes that the source code structure is stable and may produce unsatisfactory results in the case of significant code changes, such as API-breaking updates.

For example, during our evaluation of IntelliJ IDEA, we encounter four false negative samples related to *HTTPCLIENT-1803* (in Table 4). In this case, the unit tests show that the samples were patched, but Ppt4J fails to detect the patch.

Upon further examination of the bytecodes, we discover that IntelliJ IDEA imported a library of a newer version, which has been reconstructed and replaced by a different set of APIs. As a result, the code differences in the security patch were not reflected in the binaries of the new version.

## 8 RELATED WORK

**Binary similarity analysis.** The study of function similarity in binaries has been tackled through various techniques. Bourquin et al. [2] and BinDiff [5] rely on the isomorphism of control flow graphs to evaluate similarities. Khoo et al. propose Rendezvous [22], which optimizes this approach by improving the granularity of the analysis. Alternative approaches, such as BinHunt [14] and iBinHunt [24], formulate the semantic equivalence of basic blocks through symbolic execution and theorem provers.

For cross-platform capability, Pewny et al. [36] extract I/O behaviors at the basic block level. In addition, Eschweiler et al. propose discovRE [9], which generates platform-independent feature vectors from basic blocks. Furthermore, new techniques have been proposed to address the issues of efficiency and scalability. Genius [10] encodes the representations of control flow graphs as graph embeddings. Other methods like Gemini [43] further utilize neural networks. Additionally, Huang et al. [16] propose BinSequence, which uses Min-hashing to filter the search space.

**Patch presence test.** The first to publicly propose and implement patch presence tests is FIBER [44]. FIBER generates binary signatures by analyzing security patches. These signatures reflect representative changes introduced by the patches and are utilized to search for the target binary file.

Subsequent papers improve and extend FIBER's approach. Dai et al. propose BScout [7] to check the existence of patches in Java binaries without generating signatures. BScout employs new techniques to bridge the gap between source code and bytecode instructions and to check patch semantics throughout the target executable file. Jiang and Zhang propose PDiff [21] to test the patch presence of images from the downstream kernel in the open source kernel domain. PDiff generates summaries for patches. Based on semantic summaries, PDiff compares the target with mainstream versions before and after applying the patch and selects the closest for evaluation. An alternative approach, Osprey [39], employs a more lightweight static analysis algorithm compared to FIBER and improves efficiency without compromising too much accuracy. Among the related work, FIBER, PDiff and Osprey test the patch presence of C/C++ binaries, while BScout and our proposed Ppt4J target Java binaries. Compared to the state of the art, the advantages of Ppt4J are mainly its superiority of capturing minor changes in patches and its ability to extract precise semantic changes from patch diffs.

## 9 CONCLUSION AND FUTURE WORK

In this paper, we aim to address the problem with patch *diffs* in existing work by highlighting semantic changes. Then we design and implement Ppt4J, a patch presence test framework targeting Java binaries, which performs accurate tests with the help of semantic changes that reflect differences in program effects. Ppt4J is systematically evaluated on a dataset with real-world vulnerabilities. The results show that Ppt4J achieves an F1 score of 98.5% while maintaining a reasonable performance. In addition, we perform an in-the-wild evaluation Ppt4J on IntelliJ IDEA, in which it achieves an accuracy of 89.7% with no false positive results. Ppt4J detects two unpatched CVEs in a third-party library within IntelliJ IDEA, and we have reported this potential problem to the vendor. In future work, one possible research attempt is to minimize the impact of significant code changes in binaries (e.g., patch backporting and version changes, as described in Section 7). This investigation may help improve the effectiveness of our proposed approach in real-world applications.

## ACKNOWLEDGMENTS

This research is supported by the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), National Natural Science Foundation of China (No. 62141222), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

# REFERENCES


[1] Oracle and/or its affiliates. [n. d.]. Java Language and Virtual Machine Specifications. Retrieved July 1, 2023 from https://docs.oracle.com/javase/specs

[2] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. 1–10.

[3] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* (2002).

[4] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 464–468. https://doi.org/10.1145/3524842.3528482

[5] The Zynamics Company. [n. d.]. BinDiff. Retrieved July 1, 2023 from https://www.zynamics.com/bindiff.html

[6] The MITRE Corporation. [n. d.]. Common Vulnerabilities and Exposures. Retrieved July 1, 2023 from https://www.cve.org

[7] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct whole patch presence test for java executables. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1147–1164.

[8] Yaniv David and Eran Yahav. 2014. Tracelet-Based Code Search in Executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 349–360. https://doi.org/10.1145/2594291.2594343

[9] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *Ndss*, Vol. 52. 58–79.

[10] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.

[11] The Apache Software Foundation. 2023. Apache Ant. Retrieved July 1, 2023 from https://ant.apache.org

[12] The Apache Software Foundation. 2023. Log4j - Apache Log4j(TM) 2. Retrieved July 1, 2023 from https://logging.apache.org/log4j/2.x

[13] The Apache Software Foundation. 2023. Maven. Retrieved July 1, 2023 from https://maven.apache.org

[14] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20-22, 2008 Proceedings 10*. Springer, 238–255.

[15] Susan Horwitz. 1990. Identifying the Semantic and Textual Differences between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/93542.93574

[16] He Huang, Amr M Youssef, and Mourad Debbabi. 2017. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 155–166.

[17] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2023. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering* 49, 1 (2023), 44–63. https://doi.org/10.1109/TSE.2022.3140868

[18] Free Software Foundation Inc. [n. d.]. Detailed Unified (Comparing and Merging Files). Retrieved Oct 11, 2023 from https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html

[19] Gradle Inc. 2023. Gradle Build Tool. Retrieved July 1, 2023 from https://gradle.org

[20] INRIA. [n. d.]. ASM. Retrieved July 1, 2023 from https://asm.ow2.io

[21] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1149–1163. https://doi.org/10.1145/3372297.3417240

[22] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 329–338.

[23] Eugene Kuleshov. 2007. Using ASM framework to implement common bytecode transformation patterns. *Proc. of the 6th AOSD, ACM Press* (2007).

[24] Jiang Ming, Meng Pan, and Debin Gao. 2013. iBinHunt: Binary hunting with inter-procedural control flow. In *Information Security and Cryptology–ICISC 2012: 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers 15*. Springer, 92–109.

[25] Audris Mockus. 2007. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 7–7.

[26] NIST. [n. d.]. National Vulnerability Database. Retrieved July 1, 2023 from https://nvd.nist.gov

[27] NVD. 2016. CVE-2016-8739. Retrieved July 1, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2016-8739

[28] NVD. 2017. CVE-2017-1000498. Retrieved July 1, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2017-1000498

[29] NVD. 2017. CVE-2017-18349. Retrieved July 1, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2017-18349

[30] NVD. 2018. CVE-2018-11797. Retrieved July 1, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2018-11797

[31] NVD. 2018. CVE-2018-1272. Retrieved July 1, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2018-1272

[32] NVD. 2018. CVE-2018-17202. Retrieved July 1, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2018-17202

[33] NVD. 2021. CVE-2021-44228. Retrieved July 1, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2021-44228

[34] VMware Inc. or its affiliates. [n. d.]. Spring Framework. Retrieved July 1, 2023 from https://spring.io/projects/spring-framework

[35] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[36] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 709–724.

[37] SecurityScorecard. 2023. Browse cve vulnerabilities by date. Retrieved July 1, 2023 from https://www.cvedetails.com/browse-by-date.php

[38] JetBrains s.r.o. [n. d.]. IntelliJ IDEA – the Leading Java and Kotlin IDE. Retrieved July 1, 2023 from https://www.jetbrains.com/idea

[39] Peiyuan Sun, Qiben Yan, Haoyi Zhou, and Jianxin Li. 2021. Osprey: A fast and accurate patch presence test framework for binaries. *Computer Communications* 173 (2021), 95–106. https://doi.org/10.1016/j.comcom.2021.03.011

[40] Wikipedia. 2023. Diff - Wikipedia. Retrieved July 1, 2023 from https://en.wikipedia.org/wiki/Diff

[41] Wikipedia. 2023. Jaccard index - Wikipedia. Retrieved July 1, 2023 from https://en.wikipedia.org/wiki/Jaccard_index

[42] Wikipedia. 2023. Syntactic sugar - Wikipedia. Retrieved July 1, 2023 from https://en.wikipedia.org/wiki/Syntactic_sugar

[43] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 363–376.

[44] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *USENIX Security Symposium*. 887–902.