PPT4J: Patch Presence Test for Java Binaries

Zhiyuan Pan[†] Xing Hu*[†] Xin Xia[‡] Xian Zhan[§] David Lo[¶] Xiaohu Yang[†]

[†]The State Key Laboratory of Blockchain and Data Security,
Zhejiang University

[‡]Software Engineering Application Technology Lab,
Huawei

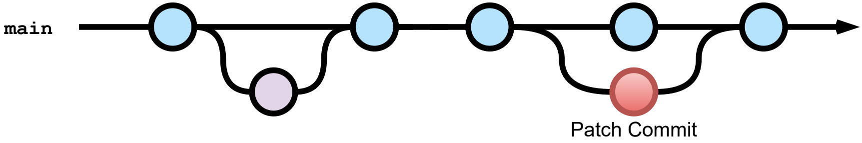
[§]The Hong Kong Polytechnic University

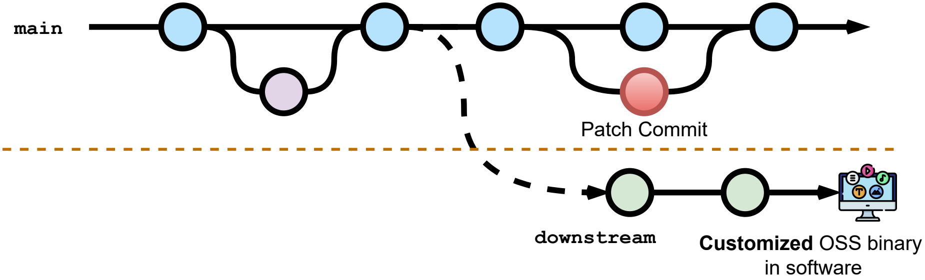
[¶]School of Computing and Information Systems,
Singapore Management University

April 17, 2024

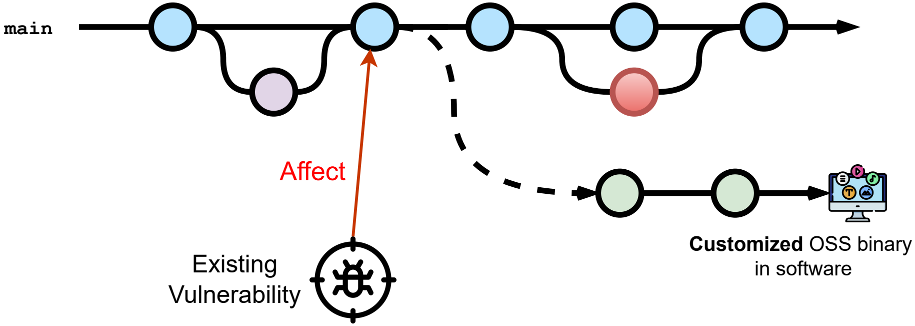
*Corresponding author

Background

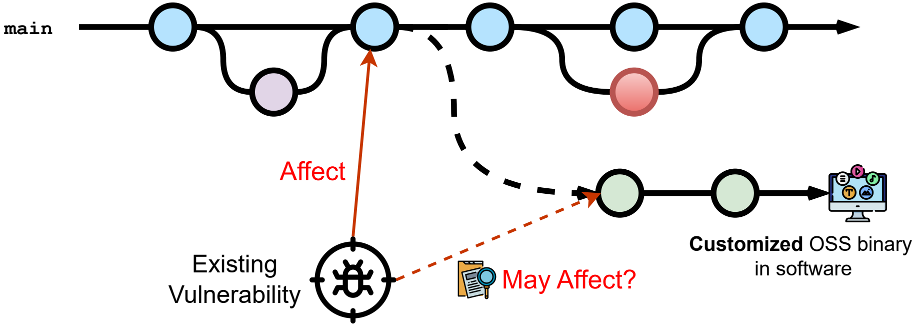


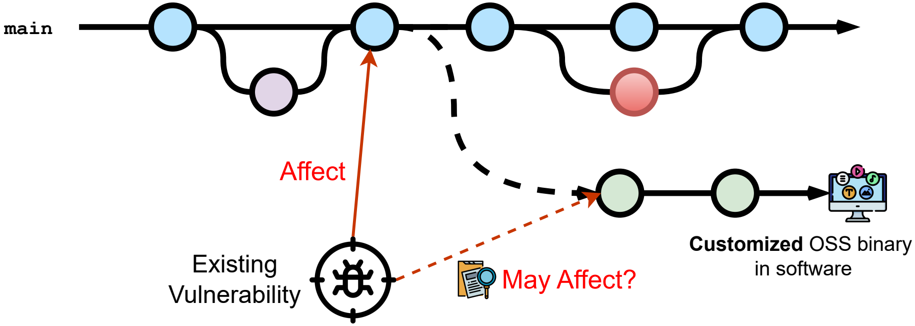

main
Patch Commit
downstream
Customized OSS binary in software

Background


main
Affect
Existing Vulnerability
Customized OSS binary in software

Background


main
Affect
Existing
Vulnerability
May Affect?
Customized OSS binary
in software



Patch Presence Test

Checks if a specific patch is applied to an unknown target binary.

Motivation

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

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

```
-    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();
+      }
```

Figure 1: Text diffs[1] $\neq$ Semantic changes

Facts

- Some (−) and (+) diff lines end up with no semantic changes.
- These lines introduce unrelated information to existing work that utilizes the complete patch diff.

[1] CVE-2017-1000498, CVE-2016-8739, and CVE-2018-11797 respectively. Diffs are simplified for illustration.

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

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

```
-    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();
+      }
```

Figure 1: Text diffs[1] $\neq$ Semantic changes

Facts

- Some (-) and (+) diff lines end up with no semantic changes.
- These lines introduce unrelated information to existing work that utilizes the complete patch diff.

Goal

- To extract precise semantic changes from diff that reflect all semantic information while not including unrelated information.

[1] CVE-2017-1000498, CVE-2016-8739, and CVE-2018-11797 respectively. Diffs are simplified for illustration.

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

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

```
-    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();
+      }
```

Figure 1: Text diffs[1] $\neq$ Semantic changes

Facts

- Some (−) and (+) diff lines end up with no semantic changes.
- These lines introduce unrelated information to existing work that utilizes the complete patch diff.

Goal

- To extract precise semantic changes from diff that reflect all semantic information while not including unrelated information.

Our Proposal

- A feature-based approach that highlights semantic changes.

[1] CVE-2017-1000498, CVE-2016-8739, and CVE-2018-11797 respectively. Diffs are simplified for illustration.

Our Approach

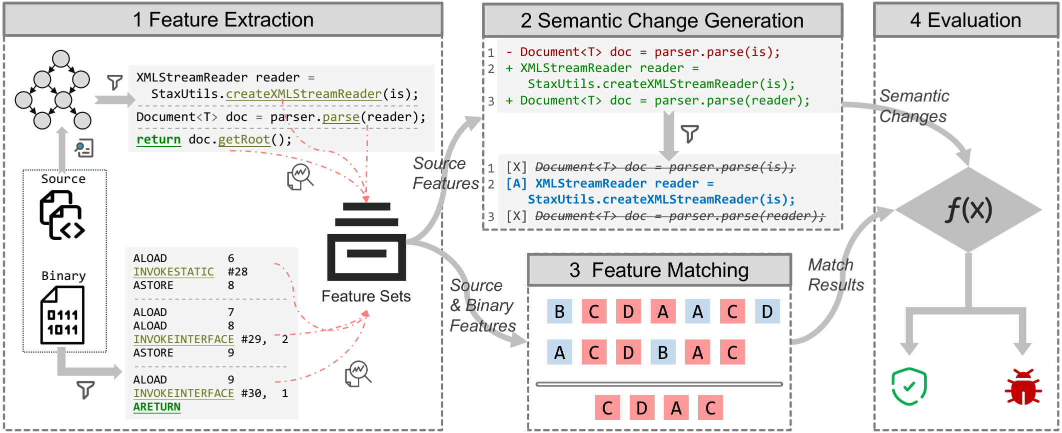


Figure 2: Overview of PPT4J

Source Code

Bytecode

Figure 3: Feature Extraction each generates a list of **unified feature sets** for source code and binary.

Feature Extraction

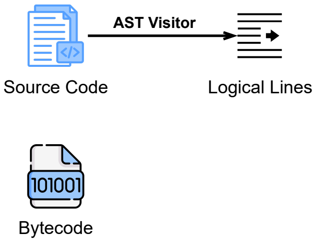


Figure 3: Feature Extraction each generates a list of **unified feature sets** for source code and binary.

Feature Extraction

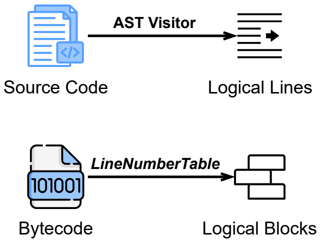


Figure 3: Feature Extraction each generates a list of **unified feature sets** for source code and binary.

Feature Extraction

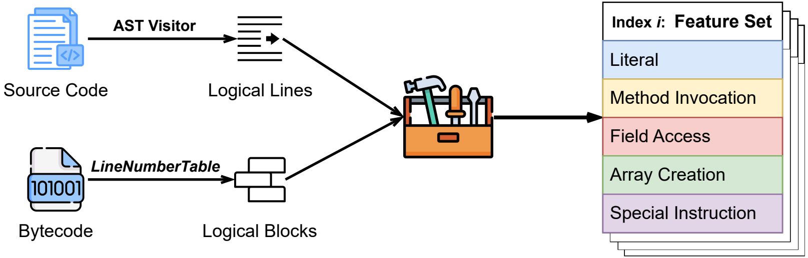


Figure 3: Feature Extraction each generates a list of **unified feature sets** for source code and binary.

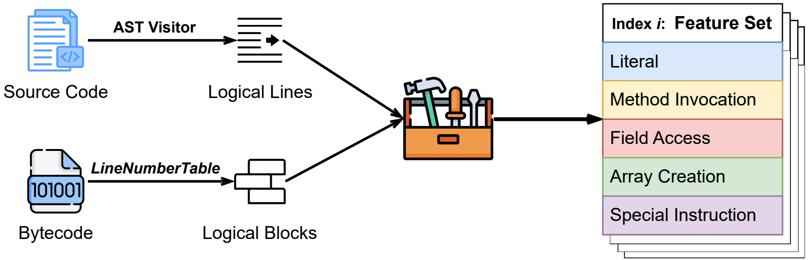


Figure 3: Feature Extraction each generates a list of **unified feature sets** for source code and binary.

- **Literal** ⟵ Extract directly & Simplify arithmetic expressions
- **Method** ⟵ Use type analysis to acquire signatures with more precise argument types

- **Special**
  1. Distinctive operators: `instanceof`, `++`/`--`, `shift`, ⋯
  2. Control flow manipulations: `return`, `throw`, `if`, `loop`[1] ⋯
  3. Syntactic sugars

[1]We analyze CFGs of bytecode to distinguish condition & loop blocks.

A sliding window-based heuristic algorithm, which utilizes **feature set similarity** to filter out semantic redundant lines.

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

1. Split diff hunks into finer-grained blocks (Type-**A**, Type-**D** or Type-**M**).
2. For each **M**-block, find the "optimal overlay" of (−) and (+) part.
3. Evaluate the similarity of feature sets within the overlay parts:
   - i. $\mathcal{J} = 1 \Rightarrow$ mark as *excluded* lines.
   - ii. $\mathcal{J} >$ threshold $\sigma_f \Rightarrow$ mark as *modification* lines.
4. Keep non-overlay parts as is, i.e., *addition* & *deletion* lines.

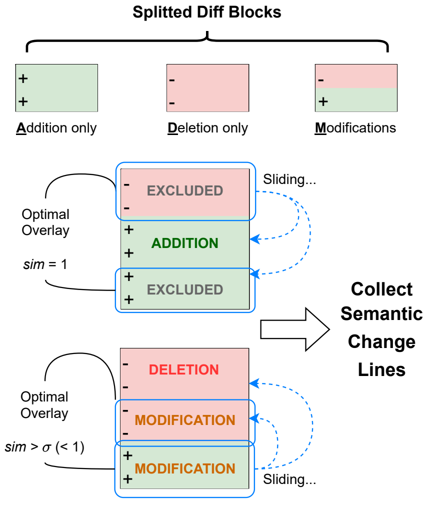


Figure 4: Examples of processing **M**-blocks

Feature Matching & Patch Presence Evaluation

What we have now:

- Lists of unified feature sets for:
  1. Reference source code before patch
  2. Reference source code after patch
  3. Target binary from user input
- Semantic change lines of the patch

It's time to figure out to what extent the binary resembles the diff part of reference sources.

1. We apply the *Longest Common Subsequence* algorithm[1] to match the feature set sequences of the binary and the source code (i.e., (i, iii) and (ii, iii)).

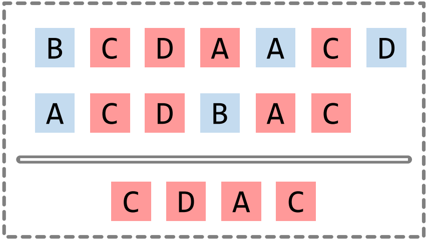


Figure 5: A simple example of the LCS algorithm. Each element in the sequence is a feature set.

[1] To make the algorithm work, we define the equivalence of two elements (i.e., feature sets) as: $\mathcal{J}(A, B) \geq \sigma_f$

Feature Matching & **Patch Presence Evaluation**

What we have now:

- Lists of unified feature sets for:
  - i Reference source code before patch
  - ii Reference source code after patch
  - iii Target binary from user input
- Semantic change lines of the patch
- **NEW:** The matching results of (i, iii) and (ii, iii)

2 Each semantic change line votes for the final result: **A**) patched; **B**) unpatched
Weighted vote: # votes = # features

- *Addition* line: if appears in Match(ii, iii), vote **A**; otherwise vote **B**
- *Deletion* line: if **not** appears in Match(i, iii), vote **A**; otherwise vote **B**
- *Modification* line pair `(pre, post)`: if "$\mathcal{J}(\text{post}, \text{binary}) > \mathcal{J}(\text{pre}, \text{binary})$", vote **A**; otherwise vote **B**
- *Excluded* lines are ignored in this procedure

Patch Support Rate $= \dfrac{\#\ \text{vote}_A}{\#\ \text{vote}}$ $\xrightarrow{\geq \sigma_p}$  $\xrightarrow{< \sigma_p}$ 

Dataset

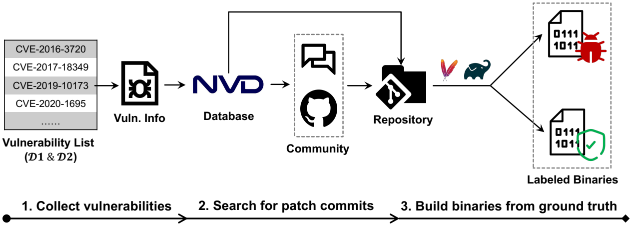


Figure 6: Steps to construct the dataset

- $\mathcal{D}1$: The Java library vulnerabilities evaluated by the baseline
- $\mathcal{D}2$: Vulnerabilities collected by Vul4J[1]

Baseline

- BScout[2] (reimplemented): A patch presence test framework specifically designed for Java binaries.

Metrics

- Accuracy
- Precision
- Recall
- F1 Score

[1]Bui et al., "Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques"
[2]Dai et al., "BScout: Direct Whole Patch Presence Test for Java Executables"

Effectiveness

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

Table 1: Test results on the dataset

Test Suite		Metrics			
		Accuracy	Precision	Recall	F1
BScout[1]	$\mathcal{D}$**1**[2]	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%**

- PPT4J does not generate false positive results.
- PPT4J outperforms the baseline BScout by 14.2% in terms of F1 score.

[1] This refers to our reimplemented version.

[2] The results on $\mathcal{D}$**1** is consistent with the original paper.

Effectiveness

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

Table 1: Test results on the dataset

Test Suite		Metrics			
		Accuracy	Precision	Recall	F1
BScout[1]	$\mathcal{D}$**1**[2]	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%**

- PPT4J does not generate false positive results.
- PPT4J outperforms the baseline BScout by 14.2% in terms of F1 score.
- PPT4J is also effective in handling patches with minor changes.

[1] This refers to our reimplemented version.
[2] The results on $\mathcal{D}$**1** is consistent with the original paper.

```
@@ -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 7: CVE-2017-18349

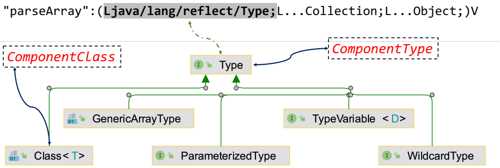


Figure 8: Class hierarchy of java.lang.reflect.Type

Case Study: 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 7: CVE-2017-18349

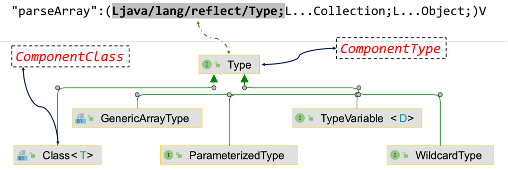


Figure 8: Class hierarchy of `java.lang.reflect.Type`

```
- parseArray(3)
       ↓
+ parseArray(3)
```

BScout ☹

V.S.

```
- parseArray(Class, ...)
       ↓
+ parseArray(Type,  ...)
```

PPT4J 😄

Efficiency

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

Table 2: Time consumption[1] 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., "BScout: Direct Whole Patch Presence Test for Java Executables".

- Most patches can be quickly analyzed.
- Time cost is not proportional to the project size because only dependent[2] bytecodes are analyzed.
- A bit slower than BScout, but the advantages in effectiveness can compensate for this.

[1] The startup time of the virtual machine and third-party dependencies is not considered.
[2] Java classes fixed by the patch, and their dependent classes.

RQ.3: How do the analyses in *Feature Extraction*[1] contribute to the overall effectiveness?

Four variants:

1. PPT4J_**FULL**: Complete version
2. PPT4J_**Δ1**: Remove type analysis
3. PPT4J_**Δ2**: Ignore special instructions (e.g., loop and branch)
4. PPT4J_**Δ3**: Remove constant propagation/folding

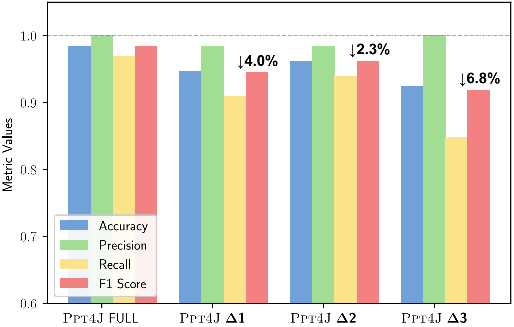


Figure 9: Test results for different variants of PPT4J

[1] Please refer to Section 3.2.3 of our paper for more details.

RQ.3: How do the analyses in *Feature Extraction*[1] contribute to the overall effectiveness?

Four variants:

1. PPT4J_**FULL**: Complete version
2. PPT4J_**Δ1**: Remove type analysis
3. PPT4J_**Δ2**: Ignore special instructions (e.g., loop and branch)
4. PPT4J_**Δ3**: Remove constant propagation/folding

The above analyses contribute to the performance improvement.

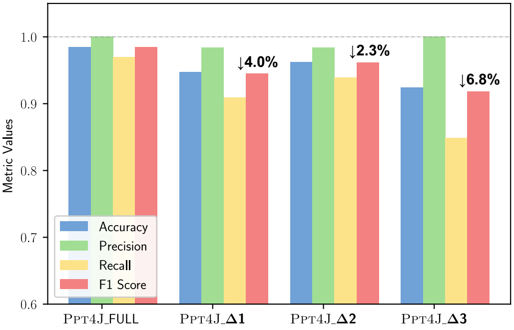


Figure 9: Test results for different variants of PPT4J

[1] Please refer to Section 3.2.3 of our paper for more details.

In-the-wild Evaluation

RQ.4: Can our approach analyze open-source libraries in real-world applications?

Test Procedures:

1. Extract OSS binaries from IntelliJ IDEA[1].
2. Run PPT[2] with specific patches.
3. Utilize unit tests in patch commits.
4. Check if **UT** results = **PPT** results.

Table 3: Results on open-source libraries within IntelliJ IDEA

	[a]	Version Timeline[b]				
		V1	V2	V3	V4	V5
CVE-2019-12402	08/19	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	TP	TP	TP	TP
CVE-2015-6420	11/15	TN	TP	TP	TP	TP

[a] Patch commit time. Retrieved from Github, in MM/YY format.
[b] 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.
[c] Info. of *CVE-Anonymous-1/2* is omitted due to "responsible reporting" principle.

[1] https://www.jetbrains.com/idea
[2] Patch Presence Test

RQ.4: Can our approach analyze open-source libraries in real-world applications?

Test Procedures:

1. Extract OSS binaries from IntelliJ IDEA[1].
2. Run PPT[2] with specific patches.
3. Utilize unit tests in patch commits.
4. Check if **UT** results = **PPT** results.

- PPT4J achieves 89.7% accuracy with no false positive results. (BScout accuracy: 76.9%, ↓ 14.3%)
- PPT4J detects two un-patched vulnerabilities. We have reported this potential problem to the vendor.

Table 3: Results on open-source libraries within IntelliJ IDEA

	[a]	Version Timeline[b]				
		V1	V2	V3	V4	V5
CVE-2019-12402	08/19	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	TP	TP	TP	TP
CVE-2015-6420	11/15	TN	TP	TP	TP	TP

[a] Patch commit time. Retrieved from Github, in MM/YY format.
[b] 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.
[c] Info. of *CVE-Anonymous-1/2* is omitted due to "responsible reporting" principle.

[1] https://www.jetbrains.com/idea
[2] Patch Presence Test

Conclusion

In summary, we made the following contributions:

1. We propose a novel patch presence test framework for Java binaries, PPT4J, which highlights semantic code differences in patches.
2. We construct a dataset to evaluate the effectiveness of PPT4J.
3. We evaluate PPT4J, with results suggesting that PPT4J outperforms the baseline and is also capable in real-world scenarios.
4. We release the replication package of PPT4J, to facilitate future research.

Thanks for your attention!

Replication Package

arXiv Preprint