BScout: Direct Whole Patch Presence Test for Java Executables

Jiarun Dai[1,¶], Yuan Zhang[1,¶], Zheyue Jiang[1], Yingtian Zhou[1], Junyan Chen[1], Xinyu Xing[2], Xiaohan Zhang[1], Xin Tan[1], Min Yang[1], and Zhemin Yang[1]

[1]*School of Computer Science, Fudan University, China*
[2]*College of Information Sciences and Technology, Pennsylvania State University, USA*
[¶]*co-first authors*

Abstract

To protect end-users and software from known vulnerabilities, it is crucial to apply security patches to affected executables timely. To this end, patch presence tests are proposed with the capability of independently investigating patch application status on a target without source code. Existing work on patch presence testing adopts a signature-based approach. To make a trade-off between the uniqueness and the stability of the signature, existing work is limited to use a small and localized patch snippet (instead of the whole patch) for signature generation, so they are inherently unreliable.

In light of this, we present BSCOUT, which directly checks the presence of a whole patch in Java executables without generating signatures. BSCOUT features several new techniques to bridge the semantic gap between source code and bytecode instructions during the testing, and accurately checks the fine-grained patch semantics in the whole target executable. We evaluate BScout with 194 CVEs from the Android framework and third-party libraries. The results show that it achieves remarkable accuracy with and without line number information (i.e., debug information) presented in a target executable. We further apply BSCOUT to perform a large-scale patch application practice study with 2,506 Android system images from 7 vendors. Our study reveals many findings that have not yet been reported.

1 Introduction

Nowadays, it is very common for software developers to borrow code from open-source projects and then integrate them into their closed-source software products. According to a recent study [17], open-source projects usually contain a large number of vulnerabilities, which could be propagated to closed-source software. To battle against n-day vulnerabilities in closed-source software, it is crucial to know whether a vulnerability has been fixed in affected closed-source software, i.e., whether a piece of software has applied a security patch for a specific vulnerability. This kind of capability, known as *patch presence test* [46], enables independent and quantitative evaluation of software security for known vulnerabilities and may urge software vendors to pay more attention to vulnerability patching. With these features, a patch presence test is an important complementary technique to protect software and end-users from known threats.

To test the presence of a patch in target software, one instinctive reaction is to perform *vulnerable code search*. However, such techniques cannot be simply applied to perform a patch presence test. First, previous work on finding vulnerable code utilizes function-level [29, 34, 42] or image-level [17, 21] code similarity to pinpoint code difference. However, such methods cannot provide sufficient granularity for patch presence test, and thus inevitably introduce high error rates in identifying patch existence. Second, existing work on finding vulnerable code primarily leverages source-to-source [29, 33, 42] or binary-to-binary [18, 22, 28] test. In the patch presence test, it requires checking patch presence in the source-to-binary fashion. Therefore, these methods are not considered as feasible solutions.

Going beyond techniques in vulnerable code search, FIBER [46] is another line of work that could be potentially used for a patch presence test. It is built to perform patch presence test for C/C++ binaries. Technically, FIBER devises a two-step approach by generating binary-level patch signatures from reference binary (built from reference source code) and then leveraging binary-to-binary test for signature matching on the target binary. To make a trade-off between the uniqueness (i.e. the signature only exists in the patch itself) and the stability (i.e. the signature is robust to benign evolution of the codebase) of the signature, FIBER only regards a small and localized part of the patch for signature generation. An obvious limitation of this approach is that it does not reflect the presence of the whole patch, thus it is inherently unreliable. Furthermore, FIBER adopts an exact signature matching mechanism on the target, which is hard to tolerate the possible code customization on the signature part. Actually, FIBER is evaluated with only 8 binaries, but it has already reported several incorrect test results due to code customization, signature instability, etc (as confirmed in [46]). Another limitation is that it requires to build the whole project for generating binary-level signatures from source code, and requires to choose the most similar image with the test target for signature generation, which is quite inflexible. These facts hinder its adoption to test a large volume of binaries.

To address the limitations above, we argue a patch presence test tool should have three properties – robustness, high accuracy, and flexibility. By *robustness*, we mean a patch presence test should rely on the whole patch rather than a small and localized patch snippet for testing. By *high accuracy*, we mean, whether a patch is applied or not, a patch presence test should accurately report its status. By *flexibility*, we mean a patch presence test should not depend on building the reference source code of the test target which is not a fully automated process in most cases.

In this work, we propose a new patch presence test tool, BSCOUT to ensure the three properties above. Different from existing techniques, BSCOUT leverages the whole patch to directly test its presence in Java **B**ytecode from **S**ource code without generating signatures. The rationales of targeting Java executables are three-fold. First, Java executables are pervasive and ubiquitous, which have been demonstrated to have numerous n-day vulnerabilities. Second, there is no existing work on patch presence test for Java executables. Third, we observe that the semantic information carried by Java bytecode instructions may be exploited to facilitate patch presence test. To the best of our knowledge, we are the first to study patch presence testing techniques on Java executables by leveraging the whole patch for a test.

Technical challenges. BSCOUT faces several non-trivial challenges of directly performing a source-to-binary test and accurately checking the presence of a group of source code changes in an executable: 1) a security patch may only introduce tiny changes [31], and we need to establish fine-grained and accurate links for these changes between the patch and the target executable; 2) since Java source code and Java bytecode instructions are expressed at different language layers and have different formats, it is difficult to perform cross-layer code equivalence test; 3) some patch-changed lines may occur multiple times in the target executable and, therefore, it requires us not to simply check the presence of the patch lines alone; 4) a patch may consist of several types of changes (e.g., addition, deletion), and it is inappropriate to adopt a uniform test strategy for each change type.

Basic idea. The design of BSCOUT is inspired by the line-level patch generation and application practice [35], which detects tiny modification by measuring the proportion of patch lines present in the target executable. To perform line-level presence testing, BSCOUT first proposes *cross-layer line-level correlative analysis*. With this, it collects language-independent features from both source code lines and bytecode instructions. Then, it utilizes feature-based line-level similarity analysis to link one source code line to several aggregated bytecode instructions (marked as a bytecode line). For bytecode instruction aggregation, BSCOUT leverages the line number information in the target executable when it is present. Otherwise, BSCOUT adopts learning-based instruction segmentation to infer the bytecode line boundary. To reliably test the presence of some patch-changed lines that occur multiple times in the target executable, BSCOUT also performs the line-level correlative analysis on the basis of the entire functions. Following the line-level presence test, BSCOUT further proposes *patch-derived differential analysis*. With this, BSCOUT categorizes patch-changed lines into three types (addition/deletion/modification). Then, it utilizes both pre-patch source code (i.e. the source code before applying the patch) and post-patch source code (i.e. the source code after applying the patch) to accurately test the presence of each type of changes in the target executable.

Results. We evaluate BSCOUT with 194 CVEs pertaining to the Android framework and third-party Java libraries. The experiments are performed on 15 Android system images (called ROMs for short in the following), 261 Android apps, and 28 desktop/server apps. Our experiment results show that BSCOUT achieves remarkable accuracy of 100% and 96.9% with and without the line number information provided. We also observed that, when applied in a patch presence test, existing work exhibits poor performance in terms of accuracy and coverage.

Given the popularity of the Android platform and its severe fragmentation issues [1], we also apply BSCOUT to study the patch application practice with 2,506 real-world Android ROMs from 7 vendors. Through our study, we have some important findings that have not yet been verified before. First, we discover Google usually patches its own devices in a proactive manner even before releasing the vulnerabilities to the public, while other vendors apply security patches relatively slowly. Second, we find that, rather than vulnerability severity or patch complexity, code customization significantly affects the adoption of a security patch. Third, we observe that all vendors have forgotten to apply patches to affected phone models. This implies it is a challenging task to manage patches among multiple software product lines. Last but not least, we surprisingly find that, to some extent, all vendors (including Google) over-claim the security patch level in their devices. There are only about 9.4% of the ROMs correctly set the security patch level.

Use cases. Potential users of BSCOUT at least include: 1) since commercialized products (usually closed-source) may inherit vulnerabilities reported in the integrated open-source projects, third-party users of these products (e.g. government agents, enterprise users, security companies) are greatly interested in knowing the patching status of these vulnerabilities; 2) developers or security testers who may even have source code access, but may still want to perform additional checks to guarantee that they have patched all n-day vulnerabilities for their products before releasing them to the public. For all these users, BScout is very helpful for its ability to assess the patching status of products without their source code.

In summary, we make the following contributions.

- We propose BSCOUT, a new technique to examine the presence of a patch for Java executables.
- Using real-world test cases, we conduct a thorough analysis and show that BSCOUT is effective and efficient in patch presence test.
- Using BSCOUT as a tool, we conduct a large-scale study and shed light on patch application practice in the real world. Our study reveals several important and interesting findings that have not yet been uncovered.

2 Challenges and Insights

We pick the security patch for *CVE-2016-3832* [14] (an Android framework vulnerability) as an example to demonstrate the challenges in patch presence test and our insights to solve these challenges. Generally, there are two kinds of bytecode formats for Java executables: traditional stack-based Java bytecode [9] and DEX bytecode [6]. Since DEX bytecode is more comprehensible, we transform Java executables to DEX bytecode with *dx* [3]. Figure 1 shows a patch snippet with related code snippets from two Java executables. In Figure 1, smali (which is the assembly language for DEX) is used to present DEX instructions.

At first, we can find that the patch snippet in Figure 1 contains 3 addition lines (line 7, 13-14) and 1 deletion line (line 12). Line 13-14 in Figure 1(a) are actually two broken lines of a single statement, thus we use line 13 to refer both of them in the following. As reported by Li et al. [31], security patches tend to introduce fewer changes to source code than general bug fixes (marked as *Challenge-I: patch is small*). Thus, to reliably check the patch presence, we need to consider all meaningful patch changes. Specifically, we need to check that whether corresponding bytecode instructions could be found in the target for every patch-changed line.

However, since Java source code and smali instructions are expressed in different languages, it is not straightforward to judge whether a statement in source code is equivalent to several smali instructions or not (marked as *Challenge-II: cross-language-layer test*). Fortunately, we observe that Java bytecode contains much semantic information. Based on this observation, we try to infer the equivalence between a Java statement and several smali instructions based on their shared semantic features. For example, line 7 in the patch snippet is a function invocation statement that invokes the "android.os.Parcel.readInt()" method and saves the function return value to a temporary variable named "userId". For this statement, we can use the name of the invoked method as a

(a) Patch Snippet for CVE-2016-3832

```
01@@ -1582,9 +1582,10 @@
02   data.enforceInterface(IActivityManager.descriptor);
03   ...
04   ...
05   int backupRestoreMode = data.readInt();
06   ...
07+  int userId = data.readInt();
08   ...
09   reply.writeNoException();
10
11@@ -2235,7 +2235,8 @@
12-  if (mActivityManager.bindBackupAgent(app, mode)) {
13+  if (mActivityManager.bindBackupAgent(app.packageName, mode,
14+      UserHandle.USER_OWNER)) {
```

(b) Smali Snippet in Target_1

```
.line 1585
invoke-virtual/range {p2 .. p2}, Landroid/os/Parcel;->readInt()I
move-result v12
 ...
.line 1587
invoke-virtual/range {p2 .. p2}, Landroid/os/Parcel;->readInt()I
move-result v13
 ...
.line 2238
:try_start_c
iget-object v6, p0, L..ActivityManagerService;->mActivityManager:L..IActivityManager;
iget-object v7, p1, L..ApplicationInfo;->packageName:Ljava/lang/String;
const/4 v8, 0x0
invoke-interface {v6, v7, p2, v8}, L..IActivityManager;->bindBackupAgent(L..String;II)Z
move-result v6
if-eqz v6, :cond_b1
```

(c) Smali Snippet in Target_2

```
.line 1584
invoke-virtual/range {p2 .. p2}, Landroid/os/Parcel;->readInt()I
move-result v12
 ...
.line 2238
:try_start_c
iget-object v6, p0, L..ActivityManagerService;->mActivityManager:L..IActivityManager;
invoke-interface {v6, p1, p2}, L..IActivityManager;->bindBackupAgent(L..ApplicationInfo;I)Z
move-result v6
if-eqz v6, :cond_b1
```

Figure 1: Patch Snippet for CVE-2016-3832 with code snippets from two target Java executables.

feature to locate corresponding smali instructions in the test target. The name of the assigned variable is not used here, because it is a temporary variable whose name is not kept after compilation. Through this feature, we can find line 1585 and line 1587 in the first target, and line 1584 in the second target can link to line 7 in the patch snippet. Similarly, we can also find all candidate smali instructions in the test target for each patch-changed line with feature-based line-to-line similarity analysis.

Based on the line-to-line link between patch lines and target bytecode instructions, we shall further judge whether the patch is present or not. For line 7 in the patch snippet, we find 2 linked lines in the first target and 1 linked line in the second target. Since line 7 is an addition line in this patch, we may simply mark both targets as patched at this time. However, we may also find that line 5 in the patch snippet (just as the same as line 7) can also link to line 1585 and line 1587 in the first target and line 1584 in the second target, while line 5 exists before the patch is applied. When we take both line 5 and line 7 in the patch snippet into account, we find that both of them have a linked line in the first target, while only one of them can have a linked line in the second target (marked as *Challenge-III: patch-changed lines may occur multiple times*). Based on this observation, it is easy to recognize the second target as unpatched and the first target as patched. The lessons we learn here are that, if patch-changed lines occur multiple times in the source code, it is hard to use the patch alone to give a reliable patch presence test result. For patch-added lines (such as line 7), we had better utilize the whole post-patch method for the test.

For line 12 and line 13 in the patch snippet, we find that they are quite similar for sharing the same feature of invoking the "android.app.IActivityManager.bindBackupAgent()" method. After line-to-line similarity analysis, both of them are linked to line 2238 in the first target and are linked to line 2238 in the second target. Because line 12 is a deletion line and line 13 is an addition line, from the perspective of line 12 we may flag both targets as unpatched, but from the perspective of line 13, we should flag both targets as patched. This contradictory result is caused by the fact that we do not recognize line 13 as a modification line on top of line 12 (marked as *Challenge-IV: patch has different types of changes*). By performing a fine-grained analysis on line 12 and line 13, we can find that line 13 invokes the method with three arguments, while line 12 invokes the method with two arguments. Considering this slight difference, we find line 2238 in the first target is more similar to line 13 than line 12 in the patch, while line 2238 in the second target is more similar to line 12 than line 13 in the patch.

Based on the checking results from line 7, line 12 and line 13, we meet a unified judgment: the executable in Figure 1(b) applies the patch, while the executable in Figure 1(c) does not apply the patch. Note that in the above example, we use the line number information in smali files to represent several smali instructions for brevity, which does not necessarily mean our tool depends on this information.

3 BScout Approach

Following the example in §2, the overall architecture of BScout is shaped in Figure 2, which consists of two steps:

Step 1: Cross-layer Line-level Correlative Analysis. Basically, it takes the pre-patch/post-patch reference source code (not the source code for the target executable) and target Java executable as input, and generate two line-to-line maps (which associate raw Java bytecode instructions to Java source code lines) between them as output. It works by first extracting cross-layer features between Java source code and Java bytecode (see details in § 3.1) and then leveraging these features to construct a line-to-line map in the scope of the whole Java method (see details in § 3.2).

Step 2: Patch-derived Differential Analysis. Based on two line-to-line maps between pre-patch/post-patch source code and target Java executable, it analyzes the fine-grained changes in the patch to guide the patch presence judgment. Specifically, it analyzes the patch to recognize not only the addition/deletion lines but also the modification lines (see details in § 3.3). Then, for each kind of patch-changed lines, it tests the presence of them in the target by comparing the match results between the target executable and the pre-patch/post-patch source code (see details in § 3.4).

3.1 Feature Extractor

Obviously, it is hard to perform equivalence tests between Java source code lines and Java bytecode instructions using existing techniques such as theorem proving [24]. Instead, we approximately test whether a Java source code line is the same with several Java bytecode instructions by measuring how many semantic features they share. To support cross-layer line-level correlative analysis between Java source code and Java bytecode, it is quite important to figure out what features to extract and how to extract.

3.1.1 Feature Set

Many types of features can be extracted from Java source code and bytecode instructions. However, not every feature is appropriate. It should meet two properties.

- *Language-independent.* A selected feature should exist in both source code and smali code. For example, temporary variable names only exist in source code. Thus, it is inappropriate for feature selection.
- *Consistently-extracted.* An appropriate feature should be extracted consistently from both source code and bytecode. For instance, we find that *array-creation* smali instructions are generated in method invocations with variable-length arguments, while there are no explicit array creations in corresponding source code. Therefore, *array-creation* is not an appropriate feature.

Ideally, all the features that fit the above two properties should be utilized. Actually, we only consider a small number of significant features that appear in common cases (as listed in Table 1). Future work could explore more features to get better performance. Even so, our prototype achieves quite good precision and recall through the evaluation (see § 4.1). In all, we consider five categories of features: *constant values*, *method invocations*, *field accesses*, *object creation* and *special instruction types*. Several features in Table 1 exclude some exceptional cases. For example, for method invocation feature, we do not consider those methods that may be generated by compilers, because these invocations may only exist in smali while do not explicitly exist in source code.

3.1.2 Feature Parser

For smali instructions and Java source code, we use separate parsers.

Parsing smali Instructions. We use *dexlib* [7] to parse smali files. All the information in a smali file can be accessed with this library, such as classes, methods, instructions, and labels. As Table 1 shows, literals are quite important features

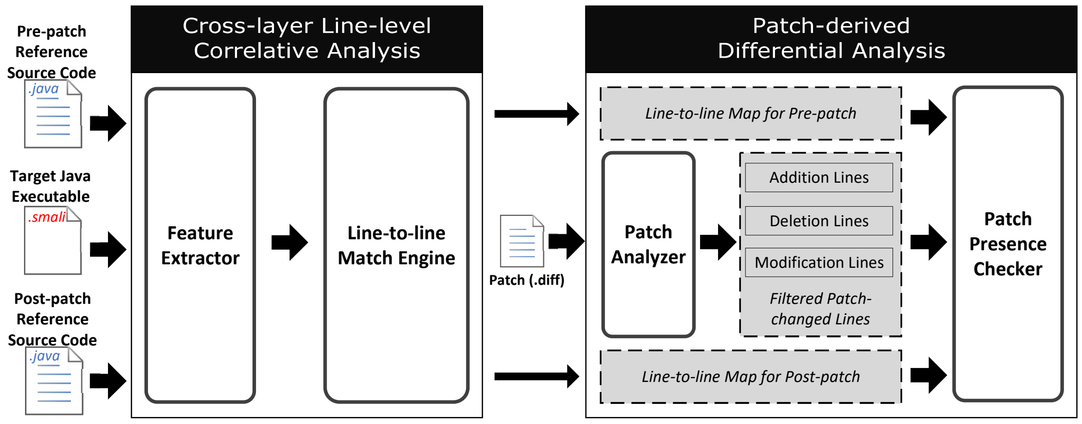
Pre-patch Reference Source Code
.java
Target Java Executable
.smali
Post-patch Reference Source Code
.java
Cross-layer Line-level Correlative Analysis
Feature Extractor
Line-to-line Match Engine
Patch (.diff)
Patch-derived Differential Analysis
Line-to-line Map for Pre-patch
Patch Analyzer
Addition Lines
Deletion Lines
Modification Lines
Filtered Patch-changed Lines
Line-to-line Map for Post-patch
Patch Presence Checker

Figure 2: Architecture of BSCOUT. There are two core techniques: *cross-layer line-level correlative analysis* and *patch-derived differential analysis*. The first technique is capable of generating a line-to-line map between a Java source method and a Java bytecode method by leveraging language-independent features, while the latter technique takes the line-to-line maps between pre-patch/post-patch reference source code (not the source code for the target executable) and target Java executable as input and utilizes the characteristics extracted from the patch itself to give a patch presence test result.

Table 1: Selected features from both Java source code and Java bytecode instructions.

Feature Category	Feature Format	Values Selected
Constant Values	Literals	String literals, integer/long numeric literals
Method Invocations	Method name with argument length	All methods except compiler-generated methods, such as *toString, valueOf, append*.
Field Access	Field name	All fields
Objection Creation	Class name	All classes except those used in compiler-generated code such as *Object, StringBuilder*
Special Instruction Type	Normalized instruction types	throw, monitor, switch, instance-of, return

in BSCOUT. In smali, literals are first loaded into virtual registers before they are used in smali instructions. Thus, we can not directly acquire the literals from operand values. To extract constant values from smali instructions, we implement constant propagation analysis. We scan the whole method to construct a table to keep all the virtual registers that have been assigned with constant values and never be overwritten by following instructions. When we want to check whether an operand register holds a constant value, we can simply look up its name in the table to get its pre-loaded constant value.

Parsing Java Source Code. Extracting semantic features from Java source code is more sophisticated than from smali instructions because Java language has complicated grammar, e.g. anonymous inner classes, nested class definitions. Meanwhile, we can neither build the source code project into executables for feature extraction, because building a project is not a fully-automated process which may require frequent manual intervention. Even for projects that use package dependency management tools such as *Maven* and *Gradle*, it is still non-trivial to set up the building environment for them.

Therefore, we have to parse Java grammars for feature extraction. After investigating several Java source code analyzers, we choose to build our feature extractor on *Spoon* [39] which is actively maintained and supports new Java features. It works by generating abstract syntax trees (AST) from Java source files, so our feature extraction procedure is implemented by traversing the ASTs. For a statement that occupies more than one textual line in the source code (e.g. line 13 and line 14 in Figure 1), we treat the split lines as a single logical line.

Literals Normalization. Literals appear as variable names (such as final static fields) in source code, but occur as constant values in disassembled smali code. An example is the variable "UserHandle.USER_OWNER" in Figure 1(a) and its constant value of "0x0" in line 2238 of Figure 1(b). To support sound feature comparison, we need to normalize literals extracted from Java source code. Our solution is to construct a global constant table by parsing all Java source code files. When we come across a variable in Java source code, we can look up the table to test whether it is a constant variable and use its constant value to construct features. Besides, we notice that Java compiler will transform string literal concatenation statements in Java source code to a concatenated string literal in smali. To correctly match these literals, we also implement this optimization during literal extraction from source code.

3.2 Line-to-line Match Engine

To perform line-level correlative analysis, we first need an oracle to test whether a source code line and several smali instructions are equivalent. Our idea is to test how many semantic features they share.

Equivalence Oracle. For convenience, we designate s and b as two feature sets extracted from a source code line and several bytecode instructions respectively. We use *Jaccard similarity* [44] between s and b to define the *equivalence oracle* as following where $T_{LineSimilarity}$ is a predefined threshold between 0 and 1. We do not use a more complicated algorithm here because *Jaccard similarity* works well enough through evaluation.

$$IsEquivalent(s,b) : Jaccard\_Sim(s,b) >= T_{LineSimilarity}$$

Notice that if a patch-changed line occurs multiple times in source code, it is unreliable to simply test its presence in smali code (see line 5 and line 7 in Figure 1 as an example). Thus, we perform a line-to-line match in the whole method scope to utilize code context. According to the presence of line number information in smali, we adopt different matching algorithms.

3.2.1 When line number information is present

Modern Java compilers such as *OpenJDK*, *Oracle JDK* and *Android SDK* all annotate line number information for compiled Java code in .class/.dex files. This information may ease the aggregation of raw smali instructions generated from the same Java source line together. Nonetheless, it is worth noting that when line number information is not present, BSCOUT can also perform an effective line-to-line match, as described in the next section.

Line Aggregation. When line number information is present in the smali file, *baksmali* [15] generates a *.line* marker with an integral line number at the first smali instruction for a source code line (see examples in Figure 1) when transforming DEX files to smali files. We split raw continuous smali instructions into blocks according to the *.line* marker. We designate a *.line* marker along with its following instructions as an aggregated line. In the following steps, BSCOUT will construct a map between Java source lines and aggregated Java bytecode lines.

After aggregating raw smali instructions into blocks according to *.line* marker, we find some exceptional cases that need to be further purified.

- *Two identical line blocks with same line number.* We find some line blocks duplicated in the smali file. For example, *baksmali* generates an identical *finally* block before each return instruction in the *try* block. Since there is only one *finally* block in the source code, we eliminate redundant *finally* blocks in the smali code and only keep one.

- *Two different line blocks with same line number.* We find that some line blocks are different but share the same line number. This is because compilers may compile a single Java statement into several line blocks. For example, a single Java *switch* statement is compiled into two-line blocks: one block starts with a *packed-switch/sparse-switch* smali instruction that indicates switch table address, and the other block starts with a leading *.switch* marker to keep the concrete switch implementation. For these blocks, we can simply merge them into one block.

Line-to-line Match. With the help of aggregated line information, precise matching can be achieved. Specifically, we sort these aggregated lines by their line numbers to facilitate the match process, and set three requirements to meet for the match results: first, each source code line should link to at most one aggregated smali line and vice versa; second, a source code line should not be matched to an aggregated smali line which has a bigger line number than previous matched smali line; third, we want to match as many source code lines/aggregated smali lines as possible.

In fact, from the above description, we have transformed the problem of the line-to-line match into the classical problem of finding the *longest common subsequence*. We apply an existing optimized algorithm in finding the longest common sequence, named *Myers algorithm* [36, 41] (which is also used by git-diff command) to BSCOUT to find an optimal match between source code lines and aggregated smali lines. Based on the line-to-line match result, we can judge whether a source code line is present in a Java executable.

3.2.2 When line number information is absent

When line number information is not present in executables, it is hard to recognize the exact line boundaries in continuous smali instructions. A straightforward idea may be searching smali instructions for every Java source code line in the whole method space. However, the unrestricted search space would cause low precision and huge overhead. Fortunately, we observe that human experts have patterns to group raw bytecode instructions. With this insight, we first use machine learning to automatically group raw continuous smali instructions into segments, and then perform matching between source code lines and smali segments.

Learning-based Instruction Segmentation. In general, we treat instruction segmentation as a sequence labeling problem [37]. Specifically, we want to assign every smali instruction with one of the four labels: S (a segment with a single instruction), B (begin of a segment), M (middle of a segment) and E (end of a segment). Our training data is constructed from smali files of 23 Android ROMs and 2,064 Maven packages. We exclude those smali files without line number information, and the remaining can be automatically labeled with S/B/M/E. In all, we extract about 1 million labeled smali methods as the training set and 10 millions of labeled smali methods as the testing set. Our model is trained with Conditional Random Fields (CRFs) [30] which is a common context-sensitive algorithm for sequence labeling. More specifically, we use CRF++ [5]

```
// Case A
String msg = uri.getPath();
EventLog.writeEvent(0x534e4554, msg);

// Case B: compound statement
EventLog.writeEvent(0x534e4554, uri.getPath());
```

Figure 3: Different cases compiled to same smali code.

(an open-source implementation of CRFs) and set cost parameter to 1 and termination criterion to 0.0001 to train the model. Considering instructions have diverse formats, we also normalize the smali instructions by removing the instruction operands. Our trained model accepts raw smali instructions as input and outputs labels for them. Based on the instruction labels, we can easily group them into segments. Through the testing set, our model shows an accuracy of 91.7% in instruction labeling. As our evaluation shows (see §4), this accuracy is good enough for our tool, so we do not try other algorithms for model training.

Two-round Line-to-segment Match. Based on the segmentation results, we first perform a one-to-one match between source code lines and instruction segments using the algorithm of finding the *longest common subsequence* (the same as in § 3.2.1). However, not all source code lines can be matched with smali segments in this round due to the existence of compound statements. We give an example in Figure 3 to demonstrate this problem. In this figure, case B is a compound statement of two statements in case A. Our instruction segmentation model inclines to group the compiled smali instructions of case B into two segments. Therefore, the line-level similarity between the compound source code line and either smali segment is hard to reach the $T_{LineSimilarity}$ threshold. As a result, this kind of compound statement is not matched during the first round.

For these unmatched source code lines, we perform a second round match. Figure 4 presents the overall design. This round of match starts sequentially from the first unmatched source code line. For this unmatched source code line, we seek matching candidates in the space of all the unmatched smali instructions just after the previous matched smali segment (see the matching candidate scope for the first unmatched line in Figure 4 as an example). Specifically, for each source code line, we set up a sliding window (see Figure 4) with variable-length to enumerate all possible smali instruction sequences. We calculate the similarity between all possible sliding windows and the source code line and select the one that hits the highest similarity. If the similarity between the smali instructions in the selected sliding window and the source code line exceeds the predefined threshold ($T_{LineSimilarity}$), we mark a line-to-line match for them and eliminate the instructions in the sliding window from the following search. Similarly, we search smali instructions for remaining unmatched source code lines.

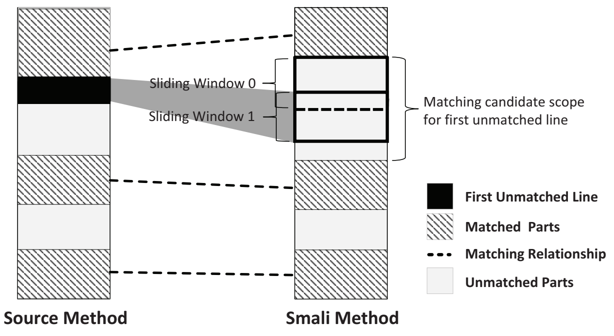

Figure 4: Second-round match for unmatched source code lines with a sliding window, which enumerates all possible matching pairs between unmatched Java source lines and smali segments to determine the best matching.

With the techniques above, BSCOUT successfully performs a line-to-line match based on the features extracted from Java source code and smali instructions, regardless of the presence of line number information. The constructed line-to-line maps will be further utilized by BSCOUT in a patch presence test.

3.3 Patch Analyzer

This module analyzes the whole patch to guide patch presence judgment. A patch is usually generated by *diff* command. As shown in Figure 1(a), a patch usually consists of multiple change blocks. Each block starts with a block header line, which indicates the position of following changes occurred in pre-patch source code and post-patch source code. From the block header, we can accurately look up each changed line and locate the affected method/class in the source code.

Not all changed lines in a patch should be considered in a patch presence test. Specifically, we perform a difference check against two kinds of changes: in-method changes and out-of-method changes. As their names indicate, out-of-method changes occur outside of method implementations (e.g. declaring a new field in a class), and in-method changes affect the concrete method implementations. Since most Java vulnerabilities are logic flaws, they should be fixed by modifying method logic. Besides, our study on 194 real-world security patches shows that nearly 80% of patch changes belong to in-method changes (as presented in Table 11 of Appendix A). Thus, we believe out-of-method changes can be ignored in testing patch presence. Under the assumption above, BSCOUT would fail on extreme cases where a patch only contains out-of-method changes. Fortunately, we do not encounter a Java patch that contains only out-of-method changes in our evaluation. Besides, we exclude code comments in the patch from the test scope.

Generally, all changes in source code can be represented as addition and deletion. However, to fully represent the patching behaviors, we further introduce modification as a new kind of code changes. Take line 12 and line 13 in Figure 1(a) as an example, the two lines actually represent a fine modification to line 12, while *diff* command generates a line to mark the deletion of the old line (line 12) and another line to mark the addition of the new line (line 13). Motivated by the example above, we adopt a heuristic-based approach to recognize modification lines: we regard adjacent deletion and addition line with similar features as a modification line. To be specific, when the similarity between two adjacent deletion and addition line is larger than the threshold $T_{LineSimilarity}$, we view these two lines as a single modification line. We also further expand the scope from one line to several consecutive lines to recognize a block of continuous modifications.

3.4 Patch Presence Checker

This module utilizes *cross-layer line-level correlative analysis* and patch analysis results to make a final judgment of patch presence. Our key idea is inspired by the lessons learned from patch presence test at the source-code level, i.e. checking how many patch-changed lines can be found in the source code. Similarly, we give patch presence test results on Java executables by checking how many patch-changed lines can be recognized in the target.

The challenge here is that there are several types of changes in a patch, we could not simply adopt a uniform presence test strategy for them. For example, to test whether an addition line is present in the test target (see line 7 in Figure 1(a)), we should compare the post-patch source code with the target bytecode, while to test a modification line (see line 12 and line 13 in Figure 1(a)), we should leverage both the pre-patch source and the post-patch source code to compare. Therefore, according to the three types (addition, deletion, and modification) of patch-changed lines, different strategies are used in the presence test.

The overall test strategy is shaped in Algorithm 1. We first use *cross-layer line-level correlative analysis* to construct two line-to-line maps between the pre-patch/post-patch source code and the target bytecode. Second, we use separate presence test strategies for addition/deletion/modification lines in a patch.

- *Line Addition.* To test the presence of an addition line, we query the line-to-line map between the post-patch source code and the target bytecode. If this line is matched, we think this line is present in the target bytecode.
- *Line Deletion.* Similarly, to test the presence of a deletion line, we query the line-to-line map between the pre-patch source code and the target bytecode. If this line is not matched, we think the target has applied this deletion.
- *Line Modification.* It is more complicated to test the presence of a modification line. We need to query both the line-to-line maps between the pre-patch/post-patch source code and the target bytecode. If this line has a higher matched similarity in the post-patch source code map than in the pre-patch source code map, we think this modification is applied in the target bytecode.

At last, a final result is given based on the presence status of every patch-changed line. Since a patch may contain multiple changed lines and we can not figure out which line is more important than another, we take all of them into account (except those filtered by *Patch Analyzer*) to make a decision. To reflect the significance of each patch-changed line, we use the number of the features extracted from each patch-changed line as its weight. Overall speaking, we calculate the patch presence ratio from the sum of the weights for all matched patch-changed lines and use a threshold ($T_{PatchPresenceRatio}$) to decide whether a patch is present or not.

Algorithm 1 Patch Presence Test

```
Input: $P_{pre}$: pre-patch source code, $P_{post}$: post-patch source code,
    $P_{smali}$: smali code, $Patch$: patch-changed lines
Output: patch presence result
 1: $Line2lineMap_{pre} \leftarrow Match(P_{pre}, P_{smali})$
 2: $Line2lineMap_{post} \leftarrow Match(P_{post}, P_{smali})$
 3: $found \leftarrow 0$
 4: $total \leftarrow 0$
 5: for each $line \in Patch$ do
 6:     $total \leftarrow total + FeaturesIn(line)$
 7:     if $isAddition(line)$ then
 8:         if $line$ is matched in $Line2lineMap_{post}$ then
 9:             $found \leftarrow found + FeaturesIn(line)$
10:         end if
11:     end if
12:     if $isDeletion(line)$ then
13:         if $line$ is not matched in $Line2lineMap_{pre}$ then
14:             $found \leftarrow found + FeaturesIn(line)$
15:         end if
16:     end if
17:     if $isModification(line)$ then
18:         $sim_{post} \leftarrow sim\_lookup(Line2lineMap_{post}, line)$
19:         $sim_{pre} \leftarrow sim\_lookup(Line2lineMap_{pre}, line)$
20:         if $sim_{post} > sim_{pre}$ then
21:             $found \leftarrow found + FeaturesIn(line)$
22:         end if
23:     end if
24: end for
25:
26: if $found/total > T_{PatchPresenceRatio}$ then
27:     return $true$
28: else
29:     return $false$
30: end if
```

4 Evaluation

We implement a prototype of BSCOUT within 9,290 LOC Java code. In detail, we utilize *Spoon* [39] as the front-end to parse Java source code, *dexlib* [7] to parse smali code, and *baksmali* [15] to transform Java executables to *smali* format for further analysis. To support traditional stack-based Java bytecode [9], BSCOUT transforms it into DEX bytecode [6] with the help of *dx* [3].

We evaluate BSCOUT with real-world programs and security patches. Though FIBER [46] is the most relevant work to BSCOUT, it only targets C/C++ binaries and it is non-trivial to make it support Java executables. Thus, we can not use it as our baseline. Actually, to the best of our knowledge, BSCOUT is the first patch presence testing tool on Java executables. Nevertheless, to illustrate the necessity of designing a dedicated tool on patch presence test for Java executables such as BSCOUT, we conduct experiments to report how two closely-related techniques behave when used on patch presence testing: *version pinning* and *function-level similarity*. It is worth noting that these techniques do not claim they are effective in patch presence testing. Here, we just want to show that the problem of patch presence testing could not be easily solved by applying existing techniques.

4.1 Results of BSCOUT

We perform experiments on two versions of BSCOUT to measure its effectiveness:

- BSCOUT which utilizes line number information (if present) in Java executables;
- BSCOUT$_\triangle$ which does not consider the line number information in Java executables (even when it is present).

This setting helps us to know the effectiveness of BSCOUT in the worst case (i.e. all line number information is stripped away). The evaluation is performed with two representative CVE datasets: *Android framework vulnerabilities* and *Java library vulnerabilities*.

4.1.1 Android Framework Vulnerabilities

Considering the popularity of Android and its severe fragmentation issues, we first use Android framework vulnerabilities to evaluate BSCOUT. In total, we randomly select 150 CVEs from Android Security Bulletin [2], ranging from August 2015 to July 2019. The patches of these CVEs are all written in Java. The affected Android versions of these CVEs are listed in in Appendix A.

Parameter Setting. Before evaluation, we need to set two parameters for BSCOUT: $T_{LineSimilarity}$ in equivalence oracle (see §3.2) and $T_{PatchPresenceRatio}$ in *Patch Presence Checker* (see §3.4). For $T_{LineSimilarity}$, we favor a low value because our line-level equivalence oracle is built on feature-based similarity, which is more coarse-grained than real semantic equivalence testing. $T_{PatchPresenceRatio}$ can make a trade-off between false positive rate (FPR) and false negative rate(FNR). In the patch presence test, we favor low FPR. To set an appropriate value for them, we build a set of ROMs from AOSP (Android Open Source Project) and label the patch status for them as ground truth. Specifically, we build all tags for all branches in AOSP and, finally, get 215 unique images. We designate this dataset as Dataset_ROM_Reference. By carefully tuning $T_{LineSimilarity}$ and $T_{PatchPresenceRatio}$, we can determine the best value of them under which BSCOUT achieves the best performance on Dataset_ROM_Reference. Finally, we set $T_{LineSimilarity}$ to 0.7 and $T_{PatchPresenceRatio}$ to 0.6.

Ground Truth. We download 15 Android ROMs from 6 vendors (marked as Dataset_ROM_GT) to measure BSCOUT. For each ROM, we unpack it and collect its affected CVEs, and manually validate the patch status for each CVE. To avoid mistakes in manual labeling, all the results are verified by two security experts. This dataset is presented in Table 2.

Results. According to the results in Table 3, the accuracy for either BSCOUT or BSCOUT$_\triangle$ is quite high, even though the test is performed directly from source code to bytecode instructions. In particular, BSCOUT achieves a remarkable accuracy of 100%. It clearly demonstrates that BSCOUT can effectively recognize fine-grained code changes at the test target by leveraging code features from both the Java source code layer and the Java bytecode layer. Besides, we note that both BSCOUT and BSCOUT$_\triangle$ exhibit no false positives. Since there is no false-negative case for BSCOUT, we manually inspect the 31 false-negative cases reported by BSCOUT$_\triangle$, and find that all these cases result in some wrong mappings between Java source code lines and smali instructions. After inspecting these mappings, we find they might be corrected by enhancing current learning-based instruction segmentation (see § 3.2.2) with the control flow-level features, and we leave it as our future work.

Efficiency of BSCOUT. We measure the test time of BSCOUT on Dataset_ROM_GT with a Windows 10 64-bit desktop computer (Intel i3-4170, 3.70GHz CPU and 12 GB memory). We run the tests one by one and collect the time cost in performing a patch presence test for each ROM-CVE pair. The detailed time cost for each ROM is presented in Table 2. Note that although a whole ROM contains millions of functions, the patches only affect a small number of functions. Thus, it is very fast for BSCOUT to locate the patch-related functions and check patch presence on them. In general, the average test time for each CVE is 0.18 seconds. Some CVEs cost more time than the average because their patches change very large methods which need more time to perform the line-to-line match. In the same way, BSCOUT$_\triangle$ is measured to have an average time cost of 13.9 seconds for each test.

4.1.2 Java Library Vulnerabilities

Since Java libraries are widely used to build applications for Android devices, desktops, servers, etc., it is also important to check whether they have patched known vulnerabilities. Thus,

Table 2: Manually-labeled Patch Presence Status for 15 Collected Android ROMs (Dataset_ROM_GT) and The Test Time for These ROMs by BSCOUT.

Model	Android Version	# of Affected CVEs[1]	# of Patched CVEs	# of Unpatched CVEs	Test Time by BSCOUT (s)	ROM Name
Google Pixel XL	7.1.2	25	7	18	8.39 / 0.34[2]	marlin-njh47d-factory-5ba1ef
Google Pixel	7.1.2	25	8	17	4.70 / 0.19	sailfish-nzh54d-factory-127f0583
Google Pixel 2 XL	8.1.0	29	13	16	1.08 / 0.04	taimen-opm4.171019.021.r1-factory-dc
Xiaomi MAX 2	7.1.1	31	15	16	6.83 / 0.22	miui_MIMAX2_7.9.14_5b67c71517_7.1
Xiaomi MAX	7.0.0	50	33	17	9.01 / 0.18	miui_MIMAX_7.9.8_5d955edf66_7.0
Xiaomi Redmi 5	8.1.0	32	32	0	3.34 / 0.10	miui_HM5_V10.3.3.0.ODACNXM_c9b6
Meizu MX5	5.1.0	9	7	2	1.53 / 0.17	MX5_6.3.0.0_cn_20180129144322
Meizu PRO 6	7.1.1	30	13	17	7.17 / 0.24	PRO_6_6.3.0.2_cn_20180327102019
Vivo X9	7.1.1	27	12	15	6.33 / 0.23	PD1616_D_7.12.7-update-full
Vivo X20	7.1.1	27	17	10	7.72 / 0.29	PD1709_A_1.16.8-update-full
Vivo NEXS	8.1.0	33	24	9	4.53 / 0.13	PD1805_A.1.23.5-update-full_15501
Oppo R11s Plus	7.1.1	33	24	9	6.91 / 0.21	R11sPlus_11_OTA_0170_all_GfK0Zhg
Oppo R9s Plus	6.0.1	62	49	13	9.37 / 0.16	R9sPlus_11_OTA_0090_all_2DQUWSz
Oppo R11s	8.1.0	34	26	8	5.44 / 0.15	R11s_11_OTA_0380_all_Q5Zf0LQ9SM
Samsung Note 9	8.1.0	27	17	10	3.78 / 0.13	LRA-N960U1UES1ARH6-20180922125

[1] Note that the number of affected CVEs may be different for ROMs with the same Android version, because vendors may remove some unwanted modules during customization.
[2] 8.39 means the total test time, while 0.34 means the average test time.

Table 3: Effectiveness Results of BSCOUT on Dataset_ROM_GT and Dataset_Apps.

Tool	Android ROMs					Java Apps					Overall	
	TP	TN	FN	FP	Acc.	TP	TN	FN	FP	Acc.	Acc.	FPR
BSCOUT	297	177	0	0	100%	291	410	0	0	100.0%	100%	0.0%
BSCOUT$_\triangle$	266	177	31	0	93.5%	286	410	5	0	99.3%	96.9%	0.0%

we collect some real-world Java (covering Android/desktop/server platforms) apps for evaluation.

Ground Truth. To ease the ground truth construction of patch status on Android apps, we write a crawler to download 4,561 open-source apps from F-Droid (which is a repository for open-source Android apps) [8]. Through parsing the Gradle build files, we recognize all the libraries that are used by each app and then collect all the reported vulnerabilities for these libraries by querying NVD [11]. From these vulnerabilities, we randomly select 15 CVEs which affect 11 libraries. We further find that these libraries are incorporated in 261 apps. Among the 261 apps, we observe that 123 apps can also be found in Google Play and 81 apps have ProGuard enabled. Similarly, we collect 12 server apps [1] and 16 desktop apps [2] for experiments, and find that they incorporate 12 Java libraries affected by 29 CVEs. We mark these 289 (=261+12+16) apps as Dataset_Apps. For each CVE, we manually label the patch status on these apps. In all, we construct 364 and 337 App-CVE pairs for Android apps and desktop/server apps as ground truth respectively.

[1]WebSpere(70011), WebLogic(12.2.1.3.0), Atlassian Confluence(10 versions)

[2]JEB Android Decompiler(3.0, 3.1), JEB Intel Decompiler(3.1), JEB ARM Decompiler(3.1), JEB MIPS Decompiler(3.1), JEB WebAssembly Decompiler(3.1), JEB Ethereum Decompiler(3.1), IntelliJ IDEA(10 versions)

The whole library dataset and CVE dataset are presented in Table 10 of Appendix A.

Tools Setup. Due to name obfuscation, BSCOUT can not directly locate patch-changed Java methods in 9 Android apps. For these cases, we leverage existing code similarity techniques [19] to recognize patch-changed methods for further patch presence test. Note that code similarity analysis may perform well in searching similar functions from a large space, but meets constraints in patch presence test due to low precision (as evaluated in §4.3). Besides, we use the same parameter setting as § 4.1.1 here.

Results. The detailed results are presented in Table 3. Overall, both BSCOUT and BSCOUT$_\triangle$ are remarkably effective by achieving an accuracy of 100% and 96.9% respectively with no false positives. By checking the 5 false negatives incurred by BSCOUT$_\triangle$, we find they are also caused by wrong line-to-line mappings which we plan to improve in the future.

4.2 Results of Version Pinning

Version pinning tools can pinpoint the most similar executable to a given target from a set of reference executables. Though version pinning tools do not directly test patch presence, two state-of-the-art tools (OSSPolice [21] and LibScout [17]) evaluate their performance in version pinning by distinguishing patched/unpatched versions of Java executables. Therefore, we conduct experiments to measure their effectiveness in the patch presence test. Specifically, we fetch the source code of OSSPolice [13] with commit hash af09514, and the source code of LibScout [10] with commit hash 4c14ca3. Furthermore, we also update some library dependencies for them to fix issues in parsing DEX files.

Experiments Setup. Since both tools require a large set of

Table 4: Results of LibScout and OSSPolice on Dataset_ROM_GT (containing 474 ROM-CVE pairs).

Tool	**Cannot Give Results**		**Can Give Results**					
	Count	Ratio	TP	TN	FP	FN	Acc.	FPR
LibScout	455	96.0%	12	1	0	6	68.4%	0%
OSSPolice	5	1.1%	69	168	6	226	50.5%	3.5%

reference images to pinpoint, we only apply OSSPolice and LibScout on Dataset_ROM_GT to ease experiment preparation. Specifically, we leverage the Dataset_ROM_Reference (consisting of 215 unique ROMs) in §4.1.1 as the reference set. Meanwhile, we also manually label the patch status of each CVE for all executables in the reference set. For each test target, we run OSSPolice and LibScout to recognize the most similar executable(s) from the reference set and use the patch presence status of the recognized executable(s) as the result of patch presence test.

Results. Table 4 presents the results of LibScout and OSSPolice in testing patch presence on Dataset_ROM_GT (containing 474 ROM-CVE pairs). We find that LibScout can not give results for 96.0% of cases and OSSPolice can not give results for 5 cases. There are two scenarios for them to give no result: 1) no image in the reference set is found to be similar to the given target, due to the heavy code customization placed on the test target; 2) at least two images are found to be quite similar to the given target with the same similarity, but they have different patch presence status. The cause of this scenario is that the code features considered by the two tools are too coarse-grained to differentiate patch changes. In the cases that OSSPolice and LibScout could give results, their accuracy is still significantly lower than that of BSCOUT. This is mainly due to that the image-level code similarity is too coarse-grained to reliably reflect the patch presence status. Overall speaking, although version pinning tools can distinguish different versions, they are too coarse-grained to test patch presence.

4.3 Results of Function-level Similarity Test

Function-level similarity testing is frequently used to locate vulnerable function clones [23, 40, 43]. Intuitively, this line of techniques can also be applied to patch presence test by measuring whether the test target is more similar to the pre-patch reference function or the post-patch one. Hence, we also perform some experiments to report the effectiveness of leveraging function-level similarity to test patch presence. As presented in §4.2, our experiments are also conducted on Dataset_ROM_GT (containing 474 ROM-CVE pairs). Since centroid [19] is widely used on Android platform [20, 21] to calculate Java method similarity, we leverage this algorithm to measure function-level similarity in this experiment.

Experiments Setup. From the 150 CVEs in Dataset_ROM_GT, we collect 471 patch-related functions. For each function, we build both pre-patch and post-patch versions from AOSP as references. In our experiment setting, the patch status of a testing target is determined by the reference which it is more similar to (i.e. if a testing target is more similar to the pre-patched one than the post-patched one, it is unpatched; otherwise it is patched). To figure the similarity degree, we define a threshold. If the distance between two similarity scores does not exceed the threshold, we think that they have the same similarity degree, and function-level similarity testing can not give a patch presence result in this scenario. By contrast, if the distance between two similarity scores exceeds the threshold, function-level similarity testing can give a patch presence result (i.e. the patch status of the more similar version).

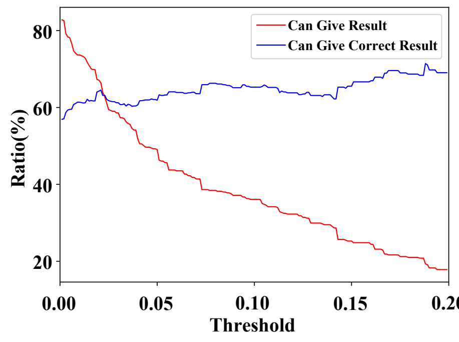

Figure 5: The ratio of cases that can give (correct) patch presence results with function-level similarity testing, by varying similarity threshold.

Results. Since the performance of function-level similarity testing is sensitive to the value of the selected similarity threshold, we vary the similarity threshold to collect testing results. More specifically, under different thresholds, we count the ROM-CVE pairs that function-level similarity testing *can give results*, and for these results, we count how many of them are correct (*can give correct results*). Figure 5 shows the results with varied similarity threshold. From this figure, we find that function-level similarity testing can at most give results for 82% of ROM-CVE pairs. For the left ROM-CVE pairs, we find that both pre-patch and post-patch reference functions have the same similarity score with the testing target. It shows that function-level similarity testing is too coarse-grained to reflect fine-grained patch changes. By increasing the similarity threshold, the ratio of *can give results* drops dramatically, because the similarity scores between testing targets and pre-patched/post-patched reference ones become more indistinguishable. Meanwhile, it is interesting to find that the ratio of *can give correct results* does not increase significantly with the increased similarity threshold. This shows that the similarity threshold does not significantly affect accuracy. The above results indicate that function-level similarity testing is not suitable for patch presence testing.

5 Empirical Study

To understand the patch application practice in the real world, we apply BScout to perform a large-scale study. Considering the severe fragmentation issues of the Android platform [1] and its wide popularity, our study is conducted on 150 collected Android framework CVEs with 2,506 ROMs collected from 7 vendors (Google, Samsung, Meizu, Xiaomi, Oppo, Vivo, and Huawei). We mark this dataset as Dataset_ROM_Large and present it in Table 5. For each ROM, we also collect several attributes (vendor, model, Android version, ROM build time, security patch level[3]) from the *build.prop* file in the ROM image. To guarantee the validity of the study, BScout is configured to leverage the line information when it is available in the testing targets. Since the presence of line information for different Java classes in a single ROM is not the same, we check the presence of this information in all CVE-related classes for all ROMs in Dataset_ROM_Large and find the ratio is 99.4%.

Our study mainly focus on three aspects of patch application practice: *patch application status*, *the lag of applying security patches*, and *the management of security patches*.

Table 5: A large-scale Dataset of ROMs Collected from Smartphone Vendors (Dataset_ROM_Large).

Vendor	Phone Models	Count	Versions	Build Time
Google	14	569	4.4.4-8.1.0	2014.06-2019.05
Samsung	24	468	5.0.0-8.1.0	2016.10-2018.09
Meizu	44	481	5.0.1-8.1.0	2015.06-2019.07
Xiaomi	45	464	4.4.4-8.1.0	2016.02-2019.08
Oppo	31	281	4.4.4-8.1.0	2014.11-2019.08
Vivo	46	152	5.0.2-8.1.0	2015.11-2019.05
Huawei	31	91	6.0.0-7.0.0	2016.01-2017.10

5.1 Patch Application Status

Ideally, when the patch for a vulnerability has been released, all ROMs built after that date should apply this patch. To measure this practice, we first recognize all the affected ROMs (marked as S_{all}) that are built after the patch release date [4] for each CVE. Thereafter, we use BScout to detect ROMs (marked as $S_{unpatched}$) from S_{all} that have not patched the corresponding CVE. To quantify the ratio of patch application status, we define the unpatched ratio for each CVE as $unpatched\_ratio = \frac{|S_{unpatched}|}{|S_{all}|}$. We find that only 9 CVEs are patched by all affected ROMs built after the patch release date, and 22 CVEs have an unpatched ratio higher than 50%, which means more than half of affected ROMs built after the patch release date are still vulnerable.

[3]Google assigns a security patch level for each public vulnerability which is actually its release date. Security patch level of a ROM indicates that this ROM has patched all the vulnerabilities released before this date.

[4]The patch release date for a CVE is its security patch level.

RQ1: Does the severity of a vulnerability affect its patch application status? It is common sense that highly severe vulnerabilities should receive more attention from vendors and are more likely to be patched by vendors to prevent potential threats. To verify whether vendors follow this practice, we correlate the unpatched ratio of each CVE to its CVSS [5] score [4], which is shown in Figure 6. We surprisingly find that the severest CVE does not have the lowest unpatched ratio. Furthermore, we perform a t-test [16] at a significance level of 0.05 to study the relationship between the unpatched ratio and the vulnerability severity. It is very interesting to find that there is no significant difference in the distribution of unpatched ratio among different CVEs under each CVSS score (except the CVSS score of 10 which has only 1 CVE) from that of the whole CVE dataset. We also verify the results among every individual vendor and confirm these observations also exist. This implies that developers may not fully aware of vulnerability severity when applying security patches, or perhaps vulnerability severity has not yet been a good indicator for developers to assess the necessity of applying security patches.

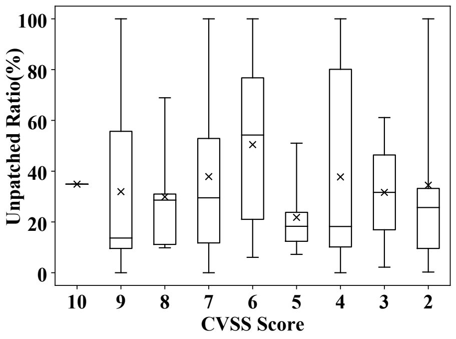

Figure 6: The unpatched ratio of all affected CVEs under different CVSS scores (RQ1).

RQ2: Does the complexity of a security patch affect its application ratio? We use the number of patch-affected lines of a patch to represent its complexity. In this way, we correlate the unpatched ratio of each CVE to its patch complexity, which is depicted in Figure 7. We perform a t-test at a significance level of 0.05 to study the relationship between the unpatched ratio and the patch complexity, and the result shows that patch complexity does not significantly affect its application ratio.

RQ3: Does code customization affect patch application? Third-party open-source code is typically customized before it is used in a software product. To figure out whether code customization becomes the obstacle to timely patching, we thoroughly analyze the relationship between the degree of code customization and the patched ratio. To measure the degree of code customization, we use function-level code similarity. To be specific, we leverage the tool introduced

[5]CVSS is a common way to assess vulnerability severity.

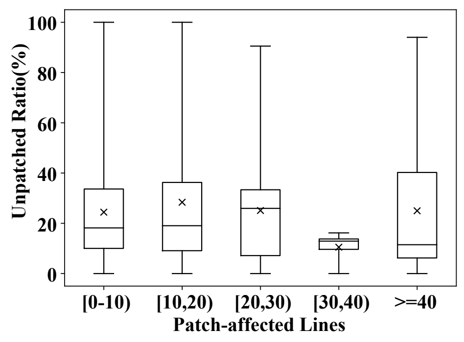

Figure 7: The correlation between unpatched ratio of all affected CVEs with the complexity of their patches (RQ2).

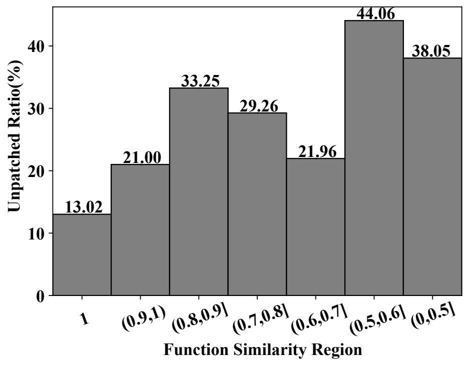

Figure 8: The unpatched ratio for all ROM-CVE pairs under different customization degrees (RQ3).

in §4.3 to calculate the centroid [19] similarity of patch-related functions between the test target and reference ones in AOSP. As shown in Figure 8, the unpatched ratio for patch-related functions with no code customization (there are 50,082 functions whose similarity score is 1 to the reference one) is significantly lower (13.02% vs 26.92%) than those with code customization (that is 44,212 functions). We also perform a one-way analysis of variance [12] to verify the significance level of the difference between the unpatched ratio of functions with and without customization, and observed the p-value is 6.64e-285. Besides, we surprisingly observed that the degree of code customization does not consistently affect the patched ratio.

Findings: *A large part of CVEs are not patched by every ROM built after the patches are released. When exploring the factors that affect the application ratio for a security patch, it seems vulnerability severity and patch complexity are not considered by vendors, but code customization is an obvious obstacle for developers in applying patches.*

5.2 The Lag of Applying Security Patches

RQ4: What is the average lag for different vendors to apply a patch? It is well-known that security patches are not applied by vendors timely, but it is difficult to estimate the lag between the patch is released and the patch is applied. Based on a large number of ROMs, we try to follow these steps to estimate such lag. First, we count the collected images in our dataset for each model and select the models with at least 10 images to study. The reason is that more collected images for a model help to track patch status more accurately. Second, we check the patch status for all the images of the selected models and select those CVEs which have been patched by at least one image for each model. Finally, we calculate the patch lag for a CVE on a model as the time span between the patch release date of the CVE and the build time of the first ROM to patch this CVE of this model. Table 6 presents the average time for the selected 99 models to patch its affected CVEs.

Table 6: The average patch lag for different vendors.

Vendor	# of Selected Phone Models	# of ROMs per Model	Average Patch Lag per Model (day)
Google	12	20 ∼ 77	-65 ∼ -21.47
Samsung	16	10 ∼ 54	38.16 ∼ 412
Xiaomi	33	10 ∼ 31	70.07 ∼ 449.25
Meizu	25	10 ∼ 29	85 ∼ 411
Vivo	2	10 ∼ 12	186.35 ∼ 194.58
Oppo	10	11 ∼ 24	62.71 ∼ 368.89
Huawei	1	10	65.55

Findings: *Google proactively patches its own devices even before announcing the vulnerabilities to the public, while third-party device manufacturers apply security patches relatively slowly. Besides, the patch lags for different phone models from the same vendor vary significantly.*

5.3 The Management of Security Patches

Vendors play an important role in applying security patches. However, it is still unknown what difficulties do vendors encounter when managing security patches. Therefore, we explore the following two research questions.

RQ5: Do vendors patch vulnerabilities for one model but ignore another? Since smartphone vendors usually manufacture several phone models, it is quite a challenging task for vendors to manage security patches among multiple software product lines. Specially, we concern whether vendors patch a known vulnerability for one model but ignore another. To study this problem, we design the following experiment. First, for each CVE and vendor, we select the ROM (marked as ROM_{first}) that is the first to apply the patch in this vendor. Second, for the same CVE and vendor, we select all models different from the model of ROM_{first}, which has a ROM not applied the patch. At last, we find all the models (called *Ill-managed Model*) that have been forgotten by the vendor to patch a vulnerability (called *Ill-managed CVE*) and this vulnerability has already been applied to some other models of the same vendor. As presented in Table 7, we find that all vendors (including Google) have ever patched a vulnerability on one model but forgot to patch the same vulnerability on another model. To further confirm whether Google has made mistakes in managing security patches, we manually check the affected ROM images and find that they indeed forgot to apply the security patches.

Table 7: Results of how often do vendors patch a vulnerability at a model while ignore another.

Vendor	# of Ill-managed CVEs	# of Ill-managed Models
Google	24	12
Samsung	76	25
Meizu	93	43
Xiaomi	75	43
Oppo	63	20
Vivo	41	35
Huawei	33	32

RQ6: Do vendors correctly set security patch level? According to Android Security Bulletin, the security patch level of a ROM indicates that the ROM has patched all the vulnerabilities released before or equal to this level. Thus, the security patch level set in a ROM is important for end-users and security experts to assess its security. However, it is unknown whether vendors correctly set security patch levels. Based on the way they set security patch levels, we consider 3 kinds of ROMs: 1) *Negligent ROM* has some vulnerabilities at a lower patch level unpatched; 2) *Diligent ROM* has patched all the vulnerabilities before its declared patch level and does not patch any vulnerability at a higher patch level; 3) *Prudent ROM* not only patches all the vulnerabilities required by its declared patch level, but also patches some vulnerabilities at a higher patch level. We label all ROMs in the dataset based on how vendors set the security patch level for them. We also exclude 233 ROMs that have not set a security patch level. The results are presented in Table 8. We surprisingly find that all vendors (including Google) have negligent ROMs. We randomly select 30 negligent ROMs from all vendors to verify the result and find our tool report correct results.

Table 8: Results for ROMs labeled according to how their vendors set security patch level.

Vendor	# of Negligent ROMs	# of Diligent ROMs	# of Prudent ROMs
Google	112	182	185
Samsung	376	12	66
Meizu	412	6	5
Xiaomi	448	2	8
Oppo	173	19	24
Vivo	139	2	6
Huawei	89	0	2

Findings: *Every vendor including Google inevitably makes mistakes in managing patches among multiple phone models, and over-claim the security patch level in some of their devices. These facts indicate that patch presence test tools such as* BSCOUT *is necessary to aid the management of security patches.*

5.4 Lessons Learned

Through our study, we find that vulnerabilities inherited from open-source projects are not actively patched by software vendors. Specifically, a large fraction of executables remain unpatched and other executables, although patched, usually suffer a long patch lag. There may be software maintenance issues inside each vendor because we find they do not timely sync security patches to all product lines and usually claim a higher security patch level than the actual one. However, we believe a fundamental cause behind these phenomenons is the lack of transparency of the patch application status or, in other words, there is no way for end-users, security companies, administrators, etc. to easily, effectively and quantitatively measure the patch application status of software. In this way, reliable, flexible and accurate patch presence tools (such as BSCOUT) are needed to urge/motivate vendors to apply security patches.

Since the resources that vendors could invest in applying security patches are always limited, they are expected to arrange the order of the patches to be applied in a rational way. However, we find no obvious clue that vendors do follow some principles to prioritize the patching process in our study. For example, the patched ratios for vulnerabilities with high severity or low patch complexity are not significantly different from those of others. Meanwhile, we observe that code customization is indeed an obstacle for vendors to apply patches, i.e. the unpatched ratio for patch-related functions with no customization is significantly lower than those with customization. Besides, even when the patch-related functions are not customized, the unpatched ratio is still as high as 13.02%. These findings indicate that more techniques are needed to help vendors apply patches, e.g. metrics to prioritize patches, back-porting security patches to lower versions and migrating security patches under code customization.

6 Limitations

BSCOUT simply ignores out-of-method changes when performing a patch presence test. Therefore, it would fail in extreme cases when patches only contain out-of-method changes. This problem could be solved by enhancing BSCOUT to also extract features from out-of-method changes. We leave this optimization as our future work.

The current implementation of BSCOUT adopts a primitive version of CRFs model to perform the learning-based instruction segmentation. Although it achieves satisfying performance, this part of the work could be further optimized by systematically evaluating all kinds of models to determine the best one.

Since BSCOUT does not know which patch lines are more important than others, it has to consider each patch line as equal importance. Besides, BSCOUT relies on users to provide the correct patch for the test target to fix the vulnerability.

Otherwise, the patch presence result could not correctly reflect the vulnerability patching status.

7 Related Work

The most related work can be categorized in three directions.

Patch Presence Test. FIBER is designed to perform a patch presence test for C/C++ binaries. Since FIBER leverages a small and localized patch snippet for exact matching, it is hard to tolerate code customization (acknowledged in §6.2 of FIBER [46]). However, we find that code customization is very common in our Dataset_ROM_GT and Dataset_ROM_Large. Thus, simply applying FIBER in these Android ROMs can not achieve satisfying performance and can not facilitate a large-scale patch application study. By leveraging the whole patch for patch presence testing, BSCOUT is more resilient to code customization and achieves remarkable accuracy.

SnoopSnitch [38] adopts a straightforward approach to perform patch presence test on native code. In its design, it enumerates all existing code commits and compilation options to prepare a large set of reference images and use the patching status of the most similar one to the testing target as the result. It is quite similar to version pining tools that are evaluated in §4.2. Obviously, this mechanism requires huge overhead in reference set preparation, bears bad scalability in testing, and is hard to tolerate code customization on the testing target.

Function-level similarity. The function-level similarity is widely used to search known buggy/vulnerable functions in a large codebase. Depending on different targets, existing work could be further divided into two classes: source code-level and binary-level. Source code-level work requires the availability of source code and usually leverage different kinds of source-level features to represent one function, such as normalized source code [29], code tokens [25, 27, 32] and parse trees [26]. This line of work differs from ours in that we do not require the source code of the test target.

Binary-level work seeks more robust features for similarity analysis. The similarity between control flow graphs is used by BinDiff [22] and BinSlayer [18] to search similar functions. Rendezvous [28] improves this technique by considering instruction mnemonics, control flow sub-graphs, and data constants. Cross-platform bug search is an appealing feature that requires to lift binary signatures to platform-independent representations. multi-MH [40] extracts high-level function semantics using the I/O behaviors at basic block level, while discovRE [43] transforms platform-dependent basic blocks into platform-independent numeric features. To improve scalability, Genius [23] converts the whole CFGs into high-level numeric feature vectors and uses graph embedding to speed up the searching process. Gemini [45] leverages neural networks to further improve the generation process of graph embedding. Centroid [19] is frequently used for calculating the similarity between Java methods. The methods mentioned above are hard to be applied to patch presence test because they extract features from the whole function rather than the patch itself, which are too coarse-grained to accurately catch the tiny changes introduced by a patch.

Version Pinning. Since patches may be applied in different versions of a library, the library version reflects patch presence status. Existing work [17, 21] collects a set of reference libraries of different versions and use similarity analysis to pinpoint a test library to the most similar one in the reference set. Specifically, OSSPolice [21] utilizes syntactic features (e.g. string constants, normalized class signatures) in a library, while LibScout [17] constructs class hierarchy profiles that are more resilient to common obfuscation techniques. Since coarse-grained features are used, existing work can differ two library versions where significant changes may occur, while works poorly to test the presence of security patches which usually introduce slight changes to the whole binary.

8 Conclusion

This paper presents BSCOUT, a tailored approach to reliably, flexibly and accurately test patch presence for Java executables. BSCOUT makes non-trivial efforts by proposing two key techniques: *cross-layer line-level correlative analysis* which utilizes feature-based line-level similarity testing to link Java source code lines to Java bytecode instructions, and *patch-derived differential analysis* which gives a reliable and precise patch presence result by calculating how many significant patch-changed lines are indeed included in the target executable. We evaluate BSCOUT with 194 CVEs from the Android framework and third-party libraries and the results show that BSCOUT is both effective and efficient. With BSCOUT, we perform an empirical study of patch application practice with 2,506 real-world Android ROMs, which reveals several interesting findings that have not been verified before and helps the community to conduct more effective efforts to fight against vulnerabilities.

Acknowledgements

We would like to thank our shepherd Martina Lindorfer and anonymous reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099, 61602123, 61602121), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

References

[1] Android fragmentation: There are now 24,000 devices from 1,300 brands. https://www.zdnet.com/article/android-fragmentation-there-are-now-24000-devices-from-1300-brands/. Accessed: 2019-08-14.

[2] Android security bulletins. https://source.android.com/security/bulletin/. Accessed: 2019-08-14.

[3] Command line tools. https://developer.android.com/studio/command-line. Accessed: 2019-08-14.

[4] Common vulnerability scoring system calculator version 2. https://nvd.nist.gov/vuln-metrics/cvss/v2-calculator. Accessed: 2019-08-14.

[5] Crf++ source code. https://github.com/taku910/crfpp. Accessed: 2019-11-08.

[6] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Accessed: 2019-08-14.

[7] Dexlib - android bytecode library. https://code.google.com/p/smali/. Accessed: 2019-08-14.

[8] F-droid - free and open source android app repository. https://f-droid.org/en/. Accessed: 2019-08-14.

[9] The java virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/. Accessed: 2019-08-14.

[10] Libscout source code. https://github.com/reddr/LibScout. Accessed: 2019-08-14.

[11] National vulnerability database. https://nvd.nist.gov. Accessed: 2019-08-14.

[12] One-way analysis of variance. https://en.wikipedia.org/wiki/One-way_analysis_of_variance. Accessed: 2019-11-08.

[13] Osspolice source code. https://github.com/osssanitizer/osspolice. Accessed: 2019-08-14.

[14] Security patch for cve-2016-3832. https://android.googlesource.com/platform/frameworks/base/+/e7cf91a198d\e995c7440b3b64352effd2e309906. Accessed: 2019-08-14.

[15] Smali/baksmali tool. https://github.com/JesusFreke/smali. Accessed: 2019-08-14.

[16] Student's t-test. https://en.wikipedia.org/wiki/Student%27s_t-test. Accessed: 2019-11-08.

[17] M. Backes, S. Bugiel, and E. Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *CCS'16*.

[18] M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *PPREW'13*.

[19] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE'14*.

[20] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds:mass vetting for new threats at the google-play scale. In *USENIX Security'15*.

[21] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *CCS'17*.

[22] T. Dullien and R. Rolles. Graph-based comparison of executable objects. *SSTIC*, 5(1):3, 2005.

[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *CCS'17*.

[24] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *ICICS'08*.

[25] J. Jang, A. Agrawal, and D. Brumley. ReDeBug - Finding Unpatched Code Clones in Entire OS Distributions. In *S&P'12*.

[26] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD - Scalable and Accurate Tree-Based Detection of Code Clones. In *ICSE'07*.

[27] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder - A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In *TSE'02*.

[28] W. M. Khoo, A. Mycroft, and R. J. Anderson. Rendezvous - a search engine for binary code. In *MSR'13*.

[29] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY - A Scalable Approach for Vulnerable Code Clone Discovery. In *S&P'17*.

[30] J. Lafferty, A. McCallum, and F. C. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[31] F. Li and V. Paxson. A large-scale empirical study of security patches. In *CCS'17*.

[32] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. In *TSE'06*.

[33] 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*.

[34] 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. In *NDSS'18*.

[35] D. MacKenzie, P. Eggert, and R. Stallman. Comparing and merging files with gnu diff and patch. *Network Theory Ltd*, 4, 2002.

[36] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1986.

[37] N. Nguyen and Y. Guo. Comparisons of sequence labeling algorithms and extensions. In *ICML'07*.

[38] K. Nohl and J. Lell. Mind the Gap: Uncovering the Android Patch Gap Through Binary-Only Patch Level Analysis. In *Hitbsecconf'2018*.

[39] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 2015.

[40] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *S&P'15*.

[41] J. Ratcliff and D. Metzener. Ratcliff-obershelp pattern recognition. *Dictionary of Algorithms and Data Structures*, 1998.

[42] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC - scaling code clone detection to big-code. In *ICSE'16*.

[43] E. Sebastian, Y. Khaled, and G. Elmar. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS'16*.

[44] P.-N. Tan et al. *Introduction to data mining*. Pearson Education India, 2006.

[45] 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*.

[46] H. Zhang and Z. Qian. Precise and accurate patch presence test for binaries. In *USENIX Security'18*.

A CVE Datasets

To evaluate the effectiveness of BSCOUT, we construct two CVE datasets. The first consists of 150 Android framework vulnerabilities collected from Android Security Bulletin spanning from August 2015 to July 2019. Table 9 gives an overview about these CVEs. The second CVE dataset has 44 vulnerabilities from 23 popular Java libraries, as shown in Table 10.

Table 9: Overview of Android Framework CVE Dataset.

Android Version	# of Affected CVEs
Android 4.*	40
Android 5.*	69
Android 6.*	95
Android 7.*	92
Android 8.*	50
Android 9.*	26
Total	150[1]

[1] Note that a CVE may affect multiple Android versions.

Table 10: Overview of Third-party Library CVE Dataset.

Library	CVE
jsoup	CVE-2015-6748
junrar	CVE-2018-12418
okhttp	CVE-2016-2402
smack	CVE-2016-10027
androidsvg	CVE-2017-1000498
google-guava	CVE-2018-10237
apache-httpclient	CVE-2013-4366
apache-jackrabbit-webdav	CVE-2016-6801
apache-commons-collections	CVE-2015-6420
apache-commons-compress	CVE-2018-1324, CVE-2018-11771
apache-commons-fileupload	CVE-2016-1000031, CVE-2016-3092 CVE-2014-0050
spring-web	CVE-2013-6429
lz4-java	CVE-2014-4715
batik-all	CVE-2018-8013, CVE-2017-5662 CVE-2015-0250
plexus-utils	CVE-2017-1000487
netty-codec-http	CVE-2015-2156, CVE-2014-0193
groovy-all	CVE-2016-6814, CVE-2015-3253
xalan-java	CVE-2014-0107
pdfbox	CVE-2016-2175
dom4j	CVE-2018-1000632
antisamy	CVE-2017-14735, CVE-2016-10006
jackson-databind	CVE-2017-7525, CVE-2017-15095 CVE-2017-17485, CVE-2018-7489
bcprov-jdk15on	CVE-2018-1000180, CVE-2016-1000352 CVE-2016-1000340, CVE-2016-1000345 CVE-2016-1000346, CVE-2016-1000341 CVE-2016-1000343, CVE-2016-1000342 CVE-2016-1000339, CVE-2015-7940 CVE-2016-1000338,

Patch Characteristics. Different to FIBER [46] which uses small and localized changes in the patch to generate binary-level signatures for patch presence test, our work advocates using the whole patch for testing. Specifically, we design *patch-derived differential analysis* to analyze the whole patch and extract features for further test. For the whole CVE dataset, we analyze their patches and present the results in Table 11. From this table, we can find BSCOUT utilizes 16.64 features in patch presence test for each CVE on average. Besides, line addition/deletion/modification are common in patches, rendering the need to leverage both pre-patch and post-patch source code for patch presence test. Meanwhile, we also find each patch has 12.14 out-of-method lines on average. Since these lines make limited contributions in fixing a vulnerability, it is necessary to recognize these lines and exclude them from the scope of patch presence test.

Table 11: Patch Characteristics for the Whole CVE Dataset (194 CVEs).

Category	Maximum	Average
# of Modified File	10	2.03
# of Modified Method	77	3.10
# of Extracted Features	117	16.64
# of In-method Addition Lines	1443	31.15
# of In-method Deletion Lines	138	11.11
# of In-method Modification Lines	14	1.24
# of Out-method Lines	806	12.14