9-2019

Automatic generation of non-intrusive updates for third-party libraries in android applications

Yue DUAN
Singapore Management University, yueduan@smu.edu.sg

Lian GAO

Jie HU

Heng YIN

Citation

DUAN, Yue; GAO, Lian; HU, Jie; and YIN, Heng. Automatic generation of non-intrusive updates for third-party libraries in android applications. (2019). *Proceedings of the 22nd International Symposium on Research on Attacks, Intrusions and Defenses, Beijing, China, Sep 23-25*.
Available at: https://ink.library.smu.edu.sg/sis_research/8140

Automatic Generation of Non-intrusive Updates for Third-Party Libraries in Android Applications

Yue Duan[1][*], Lian Gao[2], Jie Hu[2], and Heng Yin[2]

[1]Cornell University [2]University of California, Riverside
[1]yd375@cornell.edu [2]{lgao027, jhu066}@ucr.edu [2]heng@cs.ucr.edu

Abstract

Third-Party libraries, which are ubiquitous in Android apps, have exposed great security threats to end users as they rarely get timely updates from the app developers, leaving many security vulnerabilities unpatched. This issue is due to the fact that manually updating libraries can be technically non-trivial and time-consuming for app developers. In this paper, we propose a technique that performs automatic generation of non-intrusive updates for third-party libraries in Android apps. Given an Android app with an outdated library and a newer version of the library, we automatically update the old library in a way that is guaranteed to be fully backward compatible and imposes *zero* impact to the library's interactions with other components. To understand the potential impact of code changes, we propose a novel *Value-sensitive Differential Slicing* algorithm that leverages the diffing information between two versions of a library. The new slicing algorithm greatly reduces the over-conservativeness of the traditional slicing while still preserving the soundness with respect to update generation. We have implemented a prototype called LIBBANDAID. We further evaluated its efficacy on 9 popular libraries with 173 security commits across 83 different versions and 100 real-world open-source apps. The experimental results show that LIBBANDAID can achieve a high average successful updating rate of 80.6% for security vulnerabilities and an even higher rate of 94.07% when further combined with potentially patchable vulnerabilities.

1 Introduction

Third-party libraries (TPL) have been used extensively in Android to provide rich complementary functionalities for Android apps and ease the app development. This trend becomes more obvious as Android apps get increasingly complicated. Prior research has shown that every app contains 8.6 distinct TPLs on average [59], and 42.9% of apps even have more code in TPLs than in their real logic [31].

[*]This work was conducted while Yue Duan was a PhD student at University of California, Riverside, advised by Prof. Heng Yin.

Despite the benefits, TPLs can bring serious security problems for Android app. It has been revealed [15] that 70.40% of Android apps include at least one outdated TPL and 77% of the app developers only update at most a strict subset of their included TPLs, leaving many known and easy-to-exploit security vulnerabilities unpatched. In fact, updating TPLs in Android apps can be so time-consuming and tedious that developers are often forced to leave them outdated. First, updating libraries to the latest version is likely to involve considerable manual efforts to solve backward incompatibility issues [23]. Second, although 97.8% of actively used library versions with a known vulnerability could be fixed via a drop-in replacement with a specific version [23], it is impractical for app developers to manually find suitable versions for every TPL.

Existing Research. Prior efforts have been made to study and mitigate the problems with TPLs in Android apps. A variety of library detection techniques are proposed [15,21,23,31,32,38,50,59] to detect TPLs in apps and conduct measurement study. Further, techniques are proposed to isolate TPLs from the Android app. TPLs can be transformed into new processes [47,56], new apps [27,49], or new services [41]. Other works enforce in-app privilege separations [46,51] in order to keep the apps' privileges from TPLs. However, these techniques do not fix security issues per se but merely limit the harmfulness of potential problems in TPLs from the apps.

To alleviate the issues, Android patching techniques are proposed to prevent component hijacking attacks [54], detect information leakage [36,55], fix cryptographic-misuses [37] and detect runtime crashes [14]. Nonetheless, these techniques only aim to fix specific types of security issues and do not deal with the outdatedness problem on TPLs. Hence, no existing patching techniques on Android can keep TPLs updated and fix security issues in a generic fashion.

Our Approach. To solve the problem, we aim to automatically generate updates for TPLs in Android apps such that it does not require any code modification on the app side and more importantly, introduces no impact to the library interactions with other components locally and remotely as we call it *non-intrusive*. The advantages of *non-intrusiveness* are two-fold: 1). it requires zero change to the code for the given Android app so that the full backward compatibility and maintainability of the apps are ensured; 2). the internal state consistency of the app is secured since the updates guarantee no impact to the program logic of the updated library.

To achieve this goal, we need to understand the impact of the code changes between the outdated libraries and the latest versions. LIBBANDAID utilizes forward program slicing algorithm to perform Impact Analysis [18]. Traditional slicing algorithm [52] is extremely conservative and often generates unwieldy slices [17,45]. In our case, these slices will very likely to violate the *non-intrusiveness*. Techniques [44,48,58] have been proposed to prune the slices. However, they either consider only data-flow [48] or calculate relevance scores [44,58] and remove the less relevant codes. Obviously, none of them can meet our need of soundness. As a result, we propose a novel slicing algorithm called *Value-sensitive Differential Slicing* that fully leverages the diffing information between the two versions and eliminates the over-conservativeness of the traditional slicing by keeping track of value set changes for all variables. Then, we are able to produce much smaller slices while still preserving the soundness for the purpose of updates generation.

We implement a prototype called LIBBANDAID. Our system first extracts the outdated libraries from a given Android app, compares each outdated library with its latest version counterpart and generates diffing information that precisely characterizes the code changes at code statement level. Then it uses our new slicing algorithm to analyze the impact of each code change and group related changes together to form a set of candidate updates based on control and data dependencies. Finally, our system carries out a selective updating process to apply only the *non-intrusive* updates to the Android app.

We then conduct a comprehensive evaluation on LIBBANDAID by collecting 9 popular TPLs with 173 security related commits across 83 versions and 100 real world apps. The results show that LIBBANDAID can effectively patch the security vulnerabilities with a high success rate.

Contributions. In summary, this paper has made the following contributions:

- We propose an automatic non-intrusive patch generation technique and implement a prototype system called LIBBANDAID, which is the first of its kind to solve the outdatedness problem for TPLs in Android apps.
- A novel slicing algorithm called *Value-sensitive Differential Slicing* is proposed to utilize the diffing information between old and new versions of the code and reduce the over-conservativeness of the traditional forward slicing while still preserving the soundness.
- We evaluate LIBBANDAID with 9 popular TPLs with 173 security related commits across 83 different versions and 100 real world apps. The experimental results show that LIBBANDAID can effectively fix security vulnerabilities with an average success rate of 80.6% and even higher rate of 94.07% when combined with potentially patchable vulnerabilities. We demonstrate the correctness of the updated apps with automatic program testing.

2 Problem Statement

Deployment Model. Our proposed technique is anticipated to be deployed as a service for Android app developers (other than app markets or end users). Developers can feed their app that contains an outdated TPL as well as the latest version of that TPL into LIBBANDAID. It will automatically generate and apply non-intrusive updates to the TPL within the submitted app without any modification to the app's code. Our approach is designed to be conservative to guarantee a maximal updating in a non-intrusive manner. As a result, security related updates as well as other updates (e.g., new features and optimizations) can be applied to the outdated library.

It is noteworthy that the trade-off for *non-intrusiveness* is the completeness. LIBBANDAID avoids applying updates that could change the interactions among the TPL and other components. As a result, our approach makes a reasonable underlying assumption so that LIBBANDAID is designed to cover most of the security related updates.

Assumption. LIBBANDAID updates the outdated TPLs as much as possible with a high coverage for security related updates without violating the *non-intrusiveness*. The underlying assumption is that a security patch (e.g., insert a new condition check) is unlikely to introduce backward incompatibility or change how the TPL interacts with other components locally (e.g., with the app) and remotely (e.g., with TPL server). Hence, most of the security related issues can be fixed by our technique as they are very unlikely to be filtered out by the pre-defined rules that are designed to ensure the *non-intrusiveness*. This assumption is demonstrated by our evaluation with 9 most popular TPLs in Section 7.

Design Goals. LIBBANDAID achieves the following goals:

- **No source code required.** Our technique does not require any source code from Android app or the included TPLs. This is important because TPLs can be closed-source.
- **High coverage for security patches.** LIBBANDAID aims for a high coverage in updating security related issues in outdated TPLs.
- **Non-intrusiveness.** The generated updates do not change how the original app interacts with other components nor do they break the correctness of the app.

3 System Overview

In this section, we present a running example and use it to explain the work-flow of LIBBANDAID. Note that our approach works at byte-code level, source code is presented here only for ease of understanding.

3.1 Running Example

The example is based on Dropbox library [3], one of the most popular third-party libraries. Assuming that a given Android app is using Dropbox library version 3.0.3 (released in May 2017). There exist 50 commits from version 3.0.3 to the latest version 3.0.6 (released in Jan 2018), including 16 code commits [1]. Listing 1 displays two commits. Lines with colors show the code changes: lime indicates code insertions while pink and yellow specify code modifications.

The first commit is a new security feature that adds a field `accountId` in the class `DbxAuthFinish` to identify Dropbox users instead of using `userId` in older versions. The second commit is a vulnerability fix that adds a `body` field and calls `close()` function of the `body` in a callback function `onFailure()`. When Internet access is cut off, the callback function `onFailure()` will be invoked to close `body` so that potential system hang is avoided.

3.2 Overview of LibBandAid

Figure 1 delineates the overview of LibBandAid. There are four major components in LibBandAid: preprocessing, diffing analysis, update generation and selective updating.

Preprocessing. This step is to filter out the unchanged functions and generate function pairs that are modified across the two versions. Preprocessing component takes as inputs an app with outdated library and a latest version of the library, and outputs a set of function pairs. More specifically, it extracts the outdated library within the given app, analyzes all classes in the two versions of the library and performs function level byte-by-byte comparisons.

As shown in Figure 2, LibBandAid pulls out all the functions in the class and performs byte-by-byte comparisons for each function in old library with the functions in the new library as long as they share the same function name. Note that we use function name other than function signature to tolerate changes of modifier, parameter or return type. For example, `DbxAuthFinish()` in the old library is compared with `DbxAuthFinish()` and `DbxAuthFinish(String, String, Body)` in the new library. When the byte-by-byte comparison fails (two functions are not identical), we put them in the potential function mapping list and send it to diffing analysis for further analysis. This list signifies the functions in which the code changes between old and new versions reside.

Diffing Analysis. Diffing analysis in LibBandAid is to perform function level matching with a granularity of code statement so as to comprehend the exact code changes between old and new versions of a given library. To achieve this goal, we leverage the Tracelet Execution [22] and use 3-tracelet to perform code matching at code statement level. Given the output of preprocessing, 3-tracelets are generated to capture partial flow information by breaking down the control-flow graphs for each function pair. Then, the distance between tracelets are calculated to match code statements.

[1]Other non-code commits include changes in README, build file, tutorial and tests.

Listing 1: Running example

```
public class DbxAuthFinish implements CallBack {
  private String userId;
+ private String accountId;
+ private PipedRequestBody body;

- public DbxAuthFinish(String uid) {
+ public DbxAuthFinish(String uid,String aid,Body body) {
      this.userId = uid;
    + this.accountId = aid;
    + this.body = body;
  }
  public DbxAuthFinish(){
    + this.body = null;
    + this.accountId = null;
      this.userId = null;
  }
  public void onFailure (IOException ex) {
      this.error = ex;
    + if(body) this.body.close();
      notifyAll();
  }
  public DbxAuthFinish read() {
    + String accountId = null;
      String userId = null;

      while(getCurrentToken()) {
          if(n.equals("uid"))
            userId = readField();
        + else if(n.equal("accountId"))
          + accountId = readField();

        + if(accountId == null)
          + throw JsonReadexception;
      }
    - return new DbxAuthFinish(userId);
    + return new DbxAuthFinish(userId, accountId, body);
  }
+ public String getAccountId(){
  + return accountId; }
}
```

For LibBandAid, we need to further match the functions that have multiple candidates. For example, in Figure 2, `DbxAuthFinish()` in the old library can be matched to either `DbxAuthFinish()` or `DbxAuthFinish(String, String, Body)` in the new library. To understand the real change, LibBandAid leverages the distance information to further match the functions. Particularly, we consider it as a linear assignment problem and use Hungarian Algorithm [29] to find the optimal matching. Tracelet technique has demonstrated a 0.99 accuracy in comparing functions in binary code [22]. In our case, byte-code matching is easier than binary code since it is more semantic-rich. Therefore, we observe no false positive during evaluation.

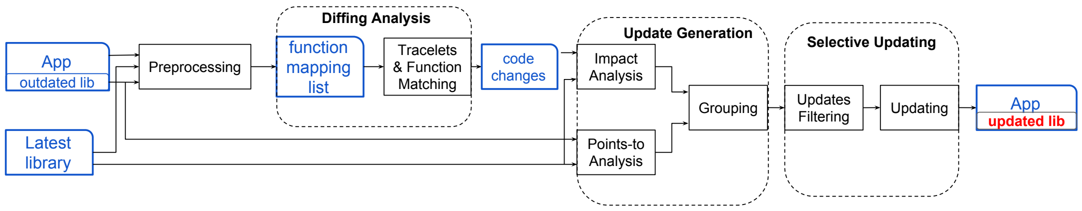


Figure 1: Architecture Overview.

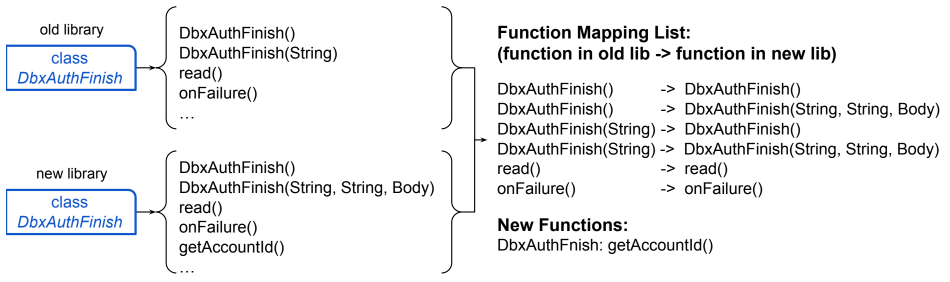

old library
class DbxAuthFinish
DbxAuthFinish()
DbxAuthFinish(String)
read()
onFailure()
...
new library
class DbxAuthFinish
DbxAuthFinish()
DbxAuthFinish(String, String, Body)
read()
onFailure()
getAccountId()
...
Function Mapping List:
(function in old lib -> function in new lib)
DbxAuthFinish() -> DbxAuthFinish()
DbxAuthFinish() -> DbxAuthFinish(String, String, Body)
DbxAuthFinish(String) -> DbxAuthFinish()
DbxAuthFinish(String) -> DbxAuthFinish(String, String, Body)
read() -> read()
onFailure() -> onFailure()
New Functions:
DbxAuthFnish: getAccountId()

Figure 2: Preprocessing.

`DbxAuthFinish()` and `DbxAuthFinish(String)` in the old library are then matched to `DbxAuthFinish()` and `DbxAuthFinish(String,String,Body)` in the new library respectively. The output of diffing analysis is the real mapping of the functions as well as a set of code changes (pairs of code statements) that precisely characterize the changes between the old and new versions of the third-party library. For our running example, the produced code changes are the same as the colored lines in Listing 1.

Update Generation. Once LIBBANDAID identifies all the code changes between the old and new versions, it starts the update generation process. The whole process takes three inputs: 1). code changes generated by diffing analysis; 2). the old version of the library; and 3). the new version of the library, and generates one output (a set of updates). It first generates system dependence graphs (SDGs) for new and old library, and then generates a slice for each code change by performing impact analysis. Finally, it performs grouping based on the alias information gathered from points-to analysis to produce updates.

The purpose of this indispensable step is two-fold. First, since many code changes have control and data dependencies with each other, LIBBANDAID should always put them together and perform updating collectively. For example, in Listing 1, Ln.10 and 13 assign values to a newly added class field `body` (defined in Ln.4). Ln.19 further calls a member function `close()` of the field. These code changes should be put into one `group` as they are the definition and usages of a same variable `body`. Second, to fulfill the non-intrusiveness design goal as described in Section 2, LIBBANDAID performs impact analysis, combines code changes with all the potentially affected code and further associates the `group` into one update so that our system can apply them as a whole if the update is indeed *non-intrusive*. As for our running example, after this step, the code changes in Listing 1 will be grouped precisely into two updates, one for each commit. More details are presented in Section 4 and 5.

Selective Updating. The last component of LIBBANDAID is selective updating. It takes the updates generated in the previous step, performs filtering to discard the ones that could potentially break the non-intrusiveness, and eventually updates the old library to generate a new app with an updated library. The core part of this step is to systematically devise a set of pre-defined rules for filtering so that the non-intrusiveness of our generated updates can be preserved. As for the running example, two updates are generated and fed into selective updating. The one related to `accountId` can potentially be filtered out since it will change an interface `DbxAuthFinish(String)` and may cause incompatibility issue. More detailed information is presented in Section 6.

4 Update Generation

In this section, we describe how LIBBANDAID performs update generation by presenting the three major steps: impact analysis, points-to analysis and grouping.

4.1 Impact Analysis

Impact Analysis is to understand the impact (affected codes) of the code changes generated from diffing analysis. Once the impact of the code changes is known, LIBBANDAID groups code changes into updates and performs filtering to remove the ones that violate the *non-intrusiveness*.

Starting from a subset of a program's behavior, program slicing technique reduces the program to a minimal form that still produces that behavior [53]. If we start slicing from a specific code change, it will conservatively includes all the codes that can potentially be affected by the change. However, traditional slicing is too conservative to be practical and tends to generate gigantic slices. The larger a slice is, the more codes it contains, hence, the bigger chance it will violate the *non-intrusiveness* and get filtered out (more in Section 6). To solve this problem, a new slicing algorithm is desired to perform a sound impact analysis with respect to our definition of *impact* while greatly reducing the over-conservativeness. We discuss the slicing in detail in Section 5.

4.2 Points-to Analysis and Grouping

After the impact analysis, LibBandAid performs points-to analysis to extract alias information and further groups code changes into updates. This step is to group slices that are accessing the same global variables or have overlapping code statements. We rely on the existing points-to analysis in Soot [1] to extract alias information.

5 Value-Sensitive Differential Slicing

In this section, we first introduce some important definitions and then describe how our slicing algorithm works in detail.

5.1 Formal Definitions

We formally define the *impact* of a code change and then lay out our definitions on the relationships between program behaviors and variable value sets, upon which the soundness of our slicing algorithm is built.

Definition 1. We denote *impact* of a code change on a code statement as $I(d,c)$, where

- d represents a code change in the new library;
- c represents a code statement that has not changed from the old to the new version of the library;

Therefore, $I(d,c) \neq \emptyset$ means that a code change d has *impact* on code statement c. Intuitively, $I(d,c) = \emptyset$ means that a code change d has no *impact* on c. We then define a code change that has no *impact* on a code statement as:

Definition 2. $I(d,c) = \emptyset \Longleftrightarrow B_c^d \subseteq B_c$, where

- B_c^d is a set of behaviors representing all possible program behaviors of c **with** d applied;
- B_c is a set of behaviors representing all possible program behaviors of c **without** applying d;

Here, the *impact* of a code change to a certain code statement is represented by the change of *program behaviors* for that code statement. If and only if all the possible program behaviors of a code statement c with the code change d applied are still within the original behavior set, we can say d has no impact on c.

We then have following definition on the relationship between value set [16] of all the variables within one code statement and the program behaviors of that code statement:

Definition 3. $VS^d(I,c) \subseteq VS(I,c) \Rightarrow B_c^d \subseteq B_c$, where

- $VS^d(I,c)$ denotes the value set of all the variables I (global and local) and their combinations used in a code statement c **with** d applied;
- $VS(I,c)$ denotes the value set of all the variables I (global and local) and their combinations used in a code statement c **without** applying d;

Essentially, this definition shows that if the value sets of all variables and their combinations used in a code statement are unchanged or a subset of the original value sets, then the program behaviors of that code statement must stay unchanged or a subset of the original ones. It gives a strong mapping from value sets of all variables in a code statement to the program behaviors of that statement. Together with Definition 2, we can draw a link between value sets of all variables in a code statement and the impact of a code change to that code statement. Specifically, our impact analysis can remove the over-conservativeness by examining the value set changes of all variables in a code statement between old and new versions of the library. If the value sets are unchanged or a subset of the original set for a statement before and after applying a code change, that means the code change has no impact on the statement and our algorithm can safely stop further slicing.

This may seem to be counter-intuitive at first glance. For example, if after applying code change d, statement c has only one behavior in its behavior set while the original behavior set has 100 behaviors, d would still be considered as having **no** impact on c as long as the one behavior is within the original behavior set. In our case, we can safely stop slicing since we know the original code c can correctly handle d and its affected behavior (it is within the original behavior set and introduces no unexpected behavior).

5.2 Basic Scheme

The core idea is to take into account the value changes of all variables between old and new versions of the code and leverage this info to reduce the over-conservativeness of the traditional slicing.

Intuitively, the basic scheme starts from a code change and performs whole library-wise context- and flow-sensitive value-set analysis (VSA) [16] on all variables and their combinations for each code statement that has dependency (control or data) with the code change. Then it compares the value sets for the variables within these code statements between two versions of the library. If there exists no change in the value sets, which means the code change has no impact on the current code statement, then our algorithm does not include that code statement in the slice. Since many values cannot be statically determined, we compute value formulas in a context- and flow-sensitive fashion as the value-set for non-constant variables.

Theoretically, this analysis is sound with respect to the definition of *impact* and could remove the over-conservativeness of traditional slicing. However, it clearly introduces a huge performance overhead for the whole library-wise context- and flow-sensitive VSA on all variables and their combinations on every control or data dependent code statement for a single code change (there could be thousands of code changes between two versions), rendering the algorithm impractical.

Consequently, we present two optimizations to this basic scheme to improve the runtime performance as well as to further reduce the over-conservativeness. Again, source code is listed just for ease of presentation while LIBBANDAID works on byte-code.

5.3 Slice-wise VSA

To reduce the complexity, we propose an optimization to narrow down the search space to the current slice which begins from the code change.

Listing 2: Slice-wise VSA

```
1  void postSingleEvent(Obj event) {
2      subscriptions = subscriptionsByEventType.
         get();
3      if (subscriptions != null
4          + && !subscriptions.isEmpty()) {
5          for (Subscription sc : subscriptions)
               {
6              postToSubscription(sc, event);
7          }
8          subscriptionFound = true;
9      }
10     ...
11 void postToSubscription(Subscription s, Obj
     event) {
12     switch (s.threadMode) {
13     case PostThread:
14         invokeSubscriber(s, event);
15     ...
```

Listing 2 shows a real-world security commit from a popular library EventBus [4]. At Ln.4, a condition check `!subscriptions.isEmpty()` is added in the new version. The traditional forward slicing will start from the code change and include every single line from Ln.4 to Ln.23 and even more codes in functions like `invokeSubscriber()` since they all have dependency with the code change. However, by manual investigation, we know the code change does not acutally introduce any new behavior to `postToSubscription()`.

For the basic scheme, we compute value sets for all variables and their combinations in every code statement that is data-dependent on the code change. For instance, for code at Ln.6, we calculate value sets for variables `sc` and `event` as well as their combinations (say, `sc` = 1 only if `event` == 0). This calculation can only be done in a whole library-wise context-sensitive fashion since the value of `event` is from the caller function `postSingleEvent()`.

To accelerate the process, we can perform VSA only within the slice instead of the whole program. This is because our analysis is to include all code statements that can be affected by the starting of the slice (a code change). That is, as long as the code change (Ln.4) does not affect the value sets of `sc` or `event` or their combinations, we could stop VSA and keep our slicing from further propagating into `postToSubscription()`. This analysis can be done much faster within the current slice other than the whole library. As a result, a much smaller slice (Ln.4-8) will be produced in a very lightweight fashion.

This optimization is an approximation to the basic scheme algorithm. It sacrifices precision of the whole library-wise VSA but greatly improves the performance. Consequently, it is more conservative than the basic scheme. For example, in a case where an assignment `a = 1` is inserted in a new library, every code that uses the variable `a` will be included under our optimization. However, a library-wise VSA may tell us that `a = 1` is still within the original value-set. Therefore, we do not need to include the code statements that are data-dependent on the newly inserted assignment.

5.4 Intra-procedural VSA

As discussed, the first optimization that searches only within the slice may bring over-conservativeness. As a result, we propose a second optimization to relax the search scope of VSA to the beginning of the function that contains the code change.

Listing 3: Intra-procedural VSA

```
1  void onResume() {
2      if (hasDropboxApp(officialAuthIntent))
3          startActivity(officialAuthIntent);
4      else
5          startWebAuth(state);
6  }
7  boolean hasDropboxApp() {
8      for (Signature sig : packInfo.sigs) {
9          - for(String dbSig : DROPBOX_SIGS)
10             - if (dbSig.equals(signature))
11                 - return true;
12
13         + if (!DROPBOX_SIGS.contains(sig)
14             + return false;
15     }
16     ...
```

Listing 3 shows another real-world security commit that fixes Android Fake ID vulnerability from Dropbox library. Code statements at Ln.9-11 in the old version are updated to codes at Ln.13-14 in the new version. Statement `return true` (Ln.11) has now become `return false` (Ln.14). Apparently, the value set of variable in the return statement has changed. According to the first optimization, our slicing algorithm will continue flowing into the call site of `hasDropboxApp()` at Ln.2, further propagate to Ln.2-5 and eventually include almost every line of code in the example.

In fact, a closer look will tell us that the code changes within `hasDropboxApp()` does not really expose any *impact* on its caller `onResume()`. Although the return value is modified, both the old and new versions of the function bear the same function-wise return value set: {`true`, `false`}. In order to capture this information, our algorithm needs to perform intra-procedural VSA beyond the scope of a slice but still within `hasDropboxApp()`, which is the function that contains the code changes. As a result, our algorithm will stop slicing and generate a much smaller slice.

From the description above, we can see that this optimization sits between the basic scheme (whole library-wise context- and flow-sensitive analysis) and the first optimization (pure slice-wise analysis). Therefore, by applying this optimization to all the variables, our slicing will be more accurate while maintaining the similar performance gain from the first optimization with negligible overhead.

5.5 *Value-sensitive Differential Slicing*

We now present the details of our slicing algorithm in Algorithm 1, which is a dependence graph based slicing algorithm as [26]. It takes three inputs and generates slice for that code change as output.

Algorithm 1 *Value-sensitive Differential Slicing*

```
1:  input1: $diff \leftarrow \{stmt_o, stmt_n\}$
2:  input2: $SDG_n \leftarrow$ {SDG of the new library.}
3:  input3: $SDG_o \leftarrow$ {SDG of the old library.}
4:  procedure $V\_Slicing(diff, SDG_n, SDG_o)$
5:      $slice \leftarrow \emptyset$
6:      $f_n \leftarrow Locate(stmt_n, SDG_n)$; $f_o \leftarrow Locate(stmt_o, SDG_o)$
7:      $workingSet \leftarrow workingSet \cup stmt_n$
8:      $slice \leftarrow slice \cup stmt_n$
9:      while $workingSet \neq \emptyset$ do
10:         $stmt \leftarrow workingSet.remove()$
11:         $Set_{succs} \leftarrow ImmediateSuccessors(stmt, SDG_n)$
12:         for $succ \in Set_{succs}$ do
13:             if $succ$ contains new invocation then
14:                 $slice \cup \leftarrow Forward\_Slicing(succ, SDG_n)$
15:             else if $succ$ is another $diff'$ then
16:                 $slice \cup \leftarrow V\_Slicing(diff', SDG_n, SDG_o)$
17:             else if $succ$ is control-dependent on $stmt$ then
18:                 $slice \leftarrow slice \cup succ$
19:                 $workingSet \leftarrow workingSet \cup succ$
20:             else if $succ$ is return statement then
21:                 if !$(RetVS(f_o) \subseteq RetVS(f_n))$ then
22:                     $slice \leftarrow slice \cup succ$
23:                     $workingSet \leftarrow workingSet \cup succ$
24:                 end if
25:             else if $succ$ is only data-dependent on $stmt$ then
26:                 $vf_n \leftarrow VSA(succ, slice, SDG_n)$
27:                 $vf_o \leftarrow VSA(succ', slice, SDG_o)$
28:                 if !$(vf_n \subseteq vf_o)$ then
29:                     $slice \leftarrow slice \cup succ$
30:                     $workingSet \leftarrow workingSet \cup succ$
31:                 end if
32:             end if
33:         end for
34:     end while
35:     Return $slice$
36: end procedure
```

The algorithm first locates the $diff$ in two SDGs (Ln.6) and adds $stmt_n$ into a $workingSet$ (Ln.7) to start the iterative process. The algorithm will continue running as long as the $workingSet$ is not empty (Ln.9). For every statement in the working set, we extract its immediate successors in SDG (Ln.11). For every immediate successor $succ$, the algorithm checks if it is another code change. There exist two cases under this scenario. First, if $succ$ is a code change that contains a new function invocation, our algorithm needs to leverage traditional slicing by calling $Forward\_Slicing()$ to keep track of the new function call (Ln.13-14) as all its codes are new codes compared to the old version. Second, if $succ$ is a normal code change, we consider it as another input to a recursive function call for $V\_Slicing()$ (Ln.15-16).

When $succ$ is not a code change, we add it into the $workingSet$ as well as the $slice$ if it is only control-dependent on $stmt$ (Ln.17-19). When $succ$ is a return statement, we apply the second optimization discussed in Section 5.4 by performing function-wise VSA for all return statements to improve the accuracy (Ln.20-23). When $succ$ is data-dependent on $stmt$, we calculate and compare the value-sets by calling $VSA()$ to extract value formulas at the scope discussed in the second optimization for both old and new versions and only add $succ$ when $stmt$ has impact on it (Ln.25-30). Eventually, it produces a slice by returning $slice$ (Ln.35).

6 Selective Updating

This component takes the generated updates from the previous step, performs filtering and applies the updates to eventually produce an updated TPL, as depicted in Figure 1.

6.1 Filtering

In this step, LIBBANDAID relies on a set of pre-defined rules to filter out the generated updates that may affect the interactions between the library and other components in order to achieve the *non-intrusiveness* goal as explained in Section 2. These rules are defined to be conservative and can guarantee that all satisfying updates will not change how the library interacts with other components. To this end, we investigate into how TPLs work and propose four categories of interactions.

Interaction with the given app. The first category is listed in the first row in Table 1. It defines the rules for interactions with the given app. When TPLs get updated by LIBBANDAID, we guarantee the interactions with the app will not be affected.

Since the interactions are always through library APIs, we need to make sure the used APIs will stay the same in terms of function names, return types, parameters and thrown exceptions. To this end, LIBBANDAID performs static program analysis to collect the library APIs used within the app and filters the updates that could change these APIs. Additionally, LIBBANDAID collects exception information and discards the updates that introduce new exceptions.

It is noteworthy that the interaction with the given app is the only category that relies on program analysis due to two reasons. First, we need to perform program analysis on the two versions of the library to understand which APIs are changed. Second, even if some APIs are indeed changed in the newer version, we may still safely update as long as the Android app does not directly call them.

Table 1: Pre-defined Rules for Filtering

Categories	Representative Behaviors		Rules
Interaction with the given app	API changes	public API signature change (return type, parameter, etc)	depend on analysis
		exception thrown change (new exception type)	depend on analysis
Interaction with server	protocol changes	incoming message change	F
		outgoing message change	F
Interaction with Android system	new Android API usages	no permission change	T
		new permission needed	F
	file manipulation	new file creation	T
		file access that modifies file pointer	F
		new file write	F
	kernel object change	thread/process creation	T
Interaction with other apps	communication to other components	new intent	F
		intent modification	F
	services	start/bind/unbind services	F

Interaction with server. Another important interaction for a TPL is to communicate with its server. For example, Dropbox library communicates with Dropbox server to access files. Therefore, our system needs to make sure that the protocol between server and client stays the same. To do so, LibBandAid scans over each update and checks if it contains code that performs network communication (incoming or outgoing). As long as such code exists, our system will conservatively choose to ignore this update. For example, if one update contains API calls such as `HttpURLConnection: getResponseMessage()`, LibBandAid will filter it out.

Interaction with system. We then consider the interactions between a TPL and the underlying Android system.

First, our update may interact with the Android framework by calling a new Android API that was not called in the old version. We rely on PScout [13] to check if the new Android API requires new Android permission. If it does, LibBandAid will discard the update. Second, we examine if an update performs any file manipulation in the Android system. Particularly, LibBandAid checks if the update affects the current system state, such as creating a new file or writing into a file. The tricky part is the file read. Our system only prevents the library from modifying the file pointer while reading a file (e.g., a call to `RandomAccessFile: seek()`). Third, library may create new kernel objects such as thread and process. LibBandAid allows this kind of interactions since they do not affect the execution of Android apps.

Interaction with other apps. The last category of interaction is the interaction with other apps in the Android system. Apps within an Android system could communicate with each other via Binder. LibBandAid disallows any update to change the communication either by creating a new intent or by changing any of the existing intent. Also, an update that starts, binds or unbinds services in the system is discarded.

6.2 Updating

After filtering out the unsatisfying updates based on our rules, LibBandAid applies the satisfying ones to the outdated library. This step is done at Jimple IR level by using byte-code rewriting capability in Soot [1]. After the rewriting, we convert the updated Jimple IR into Dalvik byte-code, repackage the DEX file with other resource files and eventually create a new Android app (APK file) with updated library.

7 Evaluation

7.1 Dataset and Configuration

We collect 9 popular Android third-party libraries [15] including Butterknife [2], Dropbox [3], EventBus [4], Glide [6], Gson [7], Leakcanary [8], Okhttp [9], Picasso [10] and Retrofit [11], with a total of 173 security commits over 83 different versions to evaluate our system. Table 2 shows the library names, total number of security commits as well as the associated library versions.

We first collect ground truth based on commit information in Github repositories to gather the vulnerability information for all the 173 security commits. Vulnerability types proposed in prior research [35] to these security related commits are presented in Table 3. As shown, our representative dataset covers a wide range of different types of vulnerabilities.

Then, we compile libraries into a number of testing versions with two requirements: 1). each testing version contains at least one security commit; 2). these testing versions cover all the security commits and version numbers that are listed in Table 2. Finally, we develop Android apps that utilize these testing versions. For each testing version other than the latest one, we feed the Android apps with these versions along with the latest version of each library into LibBandAid for evaluation. For instance, Butterknife library has 6 security commits from version 7.0.1 to 8.0.1. We compile 6 testing versions v1 to v6 to guarantee each one will contain at least 1 commit. Then we develop 5 Android apps a1 to a5 that use testing versions v1 to v5 and feed (a1,v6), (a2,v6),..,(a5,v6) into LibBandAid for experiments.

Furthermore, we collect 100 real-world Android apps from F-Droid [5] to demonstrate LibBandAid in practice. On average, the size of these apps is 4.1MB and they contain 7.1 TPLs per app. We handpick these apps since they all contain

Table 2: Overview of TPLs in Evaluation

Library	# of Security Commits	# of Testing Versions	Versions
Butterknife	6	6	7.0.1 - 8.0.1
Dropbox	11	10	3.0.0 - 3.0.6
EventBus	15	10	2.1.0 - 3.1.0
Glide	22	10	4.4.0 - 4.6.1
Gson	13	10	2.2.4 - 2.8.2
Leakcanary	42	7	1.3.1- 1.5.4
Okhttp	26	10	3.7.0 - 3.10.0
Picasso	19	10	1.5.3 - 3.0.0
Retrofit	19	10	2.0.0 - 2.4.0

at least one of the 9 libraries described above. Therefore, we can use the latest versions of these TPLs to update the apps.

7.2 Effectiveness of LIBBANDAID

As discussed, we feed each Android app that contains an older version library along with the latest version into LIBBANDAID, and then manually investigate the updated libraries to see if the commits have been updated.

Security commits can be divided into three categories: 1). 'patched'; 2). 'fail to patch'; and 3). 'potentially patchable'. 'patched' means our system can successfully update the library with the commit. 'fail to patch' shows the commits that are filtered out because of to the violation of our predefined rules. 'potentially patchable' indicates the commits that change the APIs of the library. LIBBANDAID may still update the 'potentially patchable' ones as long as the analyzed Android apps do not directly invoke the changed APIs.

By Absolute Numbers. Figure 4 gives the results in absolute numbers for the 9 libraries. The x-axis shows each execution of LIBBANDAID while y-axis is the absolute number of vulnerabilities. For example, the x-axis in Figure 4b gives the 9 executions from (a1,v10) to (a9,v10) for Dropbox library and the y-axis shows the number of security commits updated for each run. By looking at the first bar in the figure, we can see that there are total of 11 vulnerabilities between the old and new versions of the library. LIBBANDAID is able to fix 7 of them but fails in 2. Moreover, there are 2 security commits that change the APIs, so we mark them as 'potentially patchable'.

From the 9 figures, 2 libraries (Butterknife and Picasso) are shown to have no 'fail to patch' commit (no yellow bar) for all the versions. And for the rest 7 libraries, 'fail to patch' commits only take up a small average potion of total numbers across all executions. (a9,v10) execution in Okhttp (Figure 4g) is the worst case in our evaluation in which it has 1 'fail to patch' commit out of 3. Further investigation shows these commits could potentially lead to protocol changes due to the fact that Okhttp is an HTTP client and performs considerable amount of network communications. A more interesting observation is that the 'fail to patch' commits will disappear in many libraries when the outdated library becomes more recent and closer to the latest version. For Gson library in Figure 4e, starting from (a5,v10), the 'fail to patch' commit is gone.

From the experiments, LIBBANDAID could achieve an average success rate of 80.6% for updating security commits and even a higher rate of 94.07% when combining with the 'potentially patchable'.

By Vulnerability Categories. We then examine the categories of vulnerabilities that LIBBANDAID fails to update. The results are exhibited in Table 4. It shows the breakdown of vulnerabilities and the number of failures for that security commit if LIBBANDAID fails to update in all executions.

We find that among all kinds of security vulnerabilities, Info Leak is most likely to fail (1 failed in 3 total commits). In general, vulnerabilities that are related to IO exceptions and information processing (e.g., input validation, data handling) also bear relatively high failure rates. This result is expected since the updates to these vulnerabilities are most likely to affect the interactions between the library and the system or the server. Therefore, the filtering process in LIBBANDAID is triggered.

Observations. Two observations can be made from the above experimental results. First, our assumption made in Section 2 (security patches usually do not introduce backward incompatibility or change how the TPL interacts with other components) holds in practice. Second, LIBBANDAID performs better in updating relatively newer version of the library. This is because the newer the library is, the less code changes it has compared to the latest version. As a result, fewer and smaller slices will be generated and they are less likely to be filtered out by LIBBANDAID.

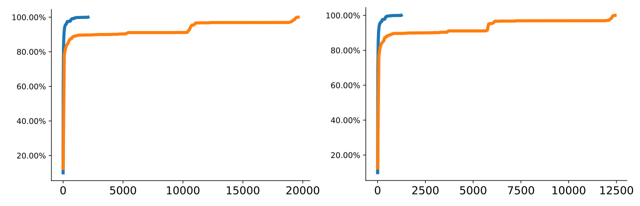
7.3 Correctness of LIBBANDAID

The correctness of LIBBANDAID is demonstrated by performing random testing as well as manual investigation for the updated apps. To this end, we first use LIBBANDAID to update TPLs within the 100 real-world apps from F-Droid [5]. Then, we collect apps with updated TPLs for testing.

For random testing, we run Monkey, which is a popular UI/Application testing tool developed by Google, on every app with an updated library for 2 hours. Although we did observe some crashes, we have confirmed that they are bugs in the original apps. No new crash is introduced by LIBBANDAID. The results demonstrate that the updated library can function normally and pass the random testing successfully. Due to the code coverage issue for random testing, we augment it with manual investigation to try out all the combinations of UI components. Combined with Monkey, our testing achieves an average code coverage of 25.7% for all the updated libraries. A closer look shows that our testing covers 30.1% of the functions that are actually updated. Admittedly, the code coverage is still far from complete, however, the correctness of LIBBANDAID can still be demonstrated to-

Table 3: Security Fixes Distribution

Vulnerability \ Library	Butterknife	Dropbox	EventBus	Glide	Gson	Leakcanary	Okhttp	Picasso	Retrofit
Improper Input Validation	1	3	3	6	5	2	7	6	1
Data Handling Error	4	4	5	3	3	3	7	1	6
Uncaught Exception	1	1	3	4	1	2	7	2	7
Memory Leak			1		1	32	1	3	
Info Leak						2			1
Race Condition			3						
Improper Access Control				2					
Uncontrolled Resource Consumption				1					
System Hang		1		1		1	2		
Uncheck Return Value				5					2
Illegal Reflective Access					1				
Stack Overflow					2			5	
Heap Access Error							1	1	1
Missing Initialization							1		1
Integer Overflow								1	
Fake ID		1							
New Security Feature		1							
Total	6	11	15	22	13	42	26	19	19



(a) CDF for number of edges (b) CDF for number of nodes

Figure 3: Effectiveness of New Slicing Algorithm

gether with our manual investigation showed in the previous Section 7.2.

7.4 Effectiveness of new slicing

Finally, we evaluate the effectiveness of *Value-sensitive Differential Slicing* by comparing it with the traditional slicing algorithm. We seek to evaluate the algorithm by answering the two following questions:

1). How well it performs in terms of over-conservativeness reduction?

2). Can it help LIBBANDAID *achieve better results?*

Over-conservativeness Reduction. We evaluate the effectiveness of *Value-sensitive Differential Slicing* by examining how much it could reduce the over-conservativeness across the 9 testing libraries. Figure 3 displays the cumulative distributions of the sizes of generated slices for traditional slicing as well as the new slicing with respect to the numbers of edges and nodes. The blue line indicates the new slicing algorithm while the yellow line represents the traditional slicing.

From the figures, we can see that *Value-sensitive Differential Slicing* could effectively reduce the number of edges as well as nodes by at least one order of magnitude. For example, 100% of the generated slices by *Value-sensitive Differential Slicing* have less than 2,500 edges and 2,000 nodes. On the contrary, traditional slicing generates way larger slices up to 20,000 edges and 12,500 nodes. This information gives us a clear view for the advantage of our slicing over the traditional slicing in terms of over-conservativeness reduction.

Updating Improvements. We further evaluate our slicing by examining the updating results improvements. The results in Section 7.2 shows LIBBANDAID could achieve a high successful updating rate for security commits when leveraging our new slicing. To evaluate, we run the experiments again with traditional slicing and compare the differences. The results show that LIBBANDAID could only achieve an updating rate of 61.84% with a rate of 74.95% when combined with the potentially patchable commits. In contrast, with the help of new slicing, our system could perform much better at rates of 80.6% and 94.07% when combined with potentially patchables as reported in Section 7.2. Detailed information is presented in Figure 6.

8 Discussion

Soundness. The soundness of our approach results from that of diffing analysis, update generation and patching respectively.

For diffing analysis, we leverage Tracelet Execution [22] technique, which has demonstrated a 0.99 accuracy in its evaluation, to compare TPLs at statement level. In our case, false positive (statements that are not code changes to be considered as changes) is impossible since we match the exact strings to confirm. Theoretically, false negatives are possible. However, we argue that false negative can only lower the successful patching rate but not bring any correctness or compatibility issue.

For update generation, the soundness of our impact analysis inherits from the soundness of traditional slicing. The basic scheme strictly follows the definition of *impact* in Section 2. However, due to the two optimizations, our slicing is still sound with respect to the definition of *impact* but may contain over-conservativeness for performance gain.

Based on the soundness analysis of our slicing, the correctness of updating is ensured by virtue of two reasons. First, LIBBANDAID introduces absolutely no code changes other than the ones from the new library itself. We assume the library developers have already tested their code before committing. Second, the completeness of each generated update is guaranteed by our slicing algorithm.

Correctness. In certain extreme cases, LIBBANDAID may affect the correctness of the updated apps in practice. For instance, a TPL function originally returns 0 only on a very rare failure, but now returns 0 for all kinds of failure after a patch. The Android app that uses the old version may simply ignore the case of returning 0 since it is so rare that developers could never make it happen during testing. However, the app may break after using LIBBANDAID.

We argue that it is the app developers' responsibility to fully test their apps in a complete fashion. But in practice, LIBBANDAID could use some lightweight sampling process such as fuzzing [60] to estimate the satisfying space for return values of a function and choose whether to perform the update. We leave this as a future work.

Limitations. To begin with, LIBBANDAID can only handle Java libraries and Java codes, and cannot update native libraries in Android apps. Moreover, non-code changes could also bring issues. For example, a version number may be stored in a file and used to communicate with server as part of the protocol. In this case, LIBBANDAID may change the protocol and introduce incompatibility. To solve the problem, we need to consider the accesses to the same file as data dependency. We also leave this as a future work.

Second, our analysis technique cannot handle obfuscated code. Recently, there is a growing tend for Android apps to use different obfuscation and packing techniques [24] to hide real logic. We argue that this is not a big problem for LIBBANDAID as it is designed for App developers who should possess unobfuscated code. Also, most of the popular TPLs [15] in Android apps are not obfuscated.

Third, our slicing relies on an accurate data dependency analysis that in turn depends on a complete modeling of Java and Android APIs. We manually write models for more than 500 most popular APIs but they still can be incomplete. This incompleteness may thwart the soundness of our analysis.

Fourth, we handle the diffing analysis as a code matching problem and leverage existing research [22] to perform analysis. We argue that this problem is orthogonal to our major focus of updating the TPLs in Android apps. We can definitely make use of the advance in code matching techniques to improve the performance of LIBBANDAID.

Finally, although LIBBANDAID analyzes the library API to collect new exception information, the analysis results in theory can be incomplete. For example, a code change in a TPL's API can call other function outside the library that eventually rises an exception. In this case, we may miss it, jeopardizing the *non-intrusiveness*.

9 Related Work

Change Impact Analysis. Change Impact Analysis [18] studies how code changes in one place could affect codes in other places of the program. Techniques have been proposed [12, 28, 30, 33, 40, 42–44, 48, 58] to improve the change impact analysis. Some of them utilize call graph analysis to study the impact of code change [12, 42, 43]. The limitation is that call graphs by nature can only provide a coarse-grained information usually at method level. Another set of research [30, 40] utilizes dynamic analysis to understand the impact of code changes. However, dynamic analysis often falls short of code coverage.

Program slicing [52] becomes a promising technique to grasp a comprehensive understanding of the impact for code changes. A series of research [28, 33, 34, 44, 48, 58] has been done towards this direction. TAILER [34] computes a tailored program that comprises the statements in all possible execution paths passing through a given statement sequence. GRACE [28] performs forward slicing to capture all potentially affected codes. To deal with the conservativeness, Sridharan et.al. [48] propose a new slicing algorithm called thin slicing that only considers value-flow. P-slicing [44] and PRIOSLICE [58] augment the forward slicing with relevance scores that indicate how likely a code statement can be affected by the change.

Android Program Patching. Automatic Program Patching in the context of Android falls into two categories: Android system patching and Android app patching. Many works have been done [19, 20, 39, 57] to perform patching on Android system and kernel. PatchDroid [39] uses in-memory patching techniques to address vulnerabilities. KARMA [20] is proposed as an adaptive live patching system for Android kernels by featuring a multi-level adaptive patching model. Embriodery [57] only targets the binary code in Android kernels by using binary rewriting techniques. It transplants official patches of known vulnerabilities to different devices by adopting heuristic matching strategies. InstaGuard [19] adopts hot-patching to patch the system programs in Android by enforcing updatable rules that contain no code to block exploits of unpatched vulnerabilities.

Android application patching techniques, on the other hand, are also proposed to mitigate security problems in Android apps. AppSealer [54], which is the most similar work with ours, performs automatic patching for preventing component hijacking attacks in Android apps. Duan et.al. [25] uses Android rewriting technique to perform privacy-preserving offloading of Android apps to the public cloud. Capper [55] and Liu et.al. [36] rewrite the Android apps to keep track of private information flow and detect privacy leakage at runtime. CDRep [37] fixes cryptographic-misuses in Android with similar byte-code rewriting technique. Azim et.al. [14] detect crashes dynamically and use byte-code rewriting technique to avoid such crashes in the future.

10 Conclusion

In this paper, we developed a novel technique named LIBBANDAID to solve the outdatedness problem for TPLs in Android apps by automatically generating non-intrusive updates. Our system extracts the outdated library within apps, compares it to the latest version of the library and generates diffing information that precisely characterizes the code changes at code statement level. Then, it analyzes the impact of each code change and generates updates. To do so, we propose a novel slicing algorithm named *Value-sensitive Differential Slicing* to reduce the over-conservativeness of the traditional slicing algorithm while still preserving the soundness. LIBBANDAID further performs selective updating by filtering out the updates that can potentially change the interactions between the library and other components. Our evaluation on 9 real-world popular third-party libraries and 100 real-world Android apps demonstrates that LIBBANDAID could effectively patch the security vulnerabilities within libraries with an average of 80.6% success rate and an even higher 94.07% when combined with potentially patchable vulnerabilities.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by DARPA under grant FA8750-16-C-0044.

References

[1] Soot: a Java Optimization Framework.

[2] Butterknife: Bind Android views and callbacks to fields and methods. https://github.com/JakeWharton/butterknife, 2018.

[3] Dropbox: A Java library for the Dropbox Core API. https://github.com/dropbox/dropbox-sdk-java/, 2018.

[4] EventBus. https://github.com/greenrobot/EventBus, 2018.

[5] F-Droid - Free and Open Source Android App Repository. https://f-droid.org/en/, 2018.

[6] Glide: An image loading and caching library. https://github.com/bumptech/glide, 2018.

[7] Gson: Java serialization library. https://github.com/google/gson, 2018.

[8] Leakcanary: A memory leak detection library for Android and Java. https://github.com/square/leakcanary, 2018.

[9] Okhttp: An HTTP+HTTP/2 client for Android and Java applications. https://github.com/square/okhttp, 2018.

[10] Picasso: A powerful image downloading and caching library for Android. https://github.com/square/picasso, 2018.

[11] Retrofit: Type-safe HTTP client. https://github.com/square/retrofit, 2018.

[12] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, pages 432–441. ACM, 2005.

[13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[14] Md Tanzirul Azim, Iulian Neamtiu, and Lisa M Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 623–628. ACM, 2014.

[15] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[16] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *International Conference on Compiler Construction*, pages 250–254. Springer, 2005.

[17] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8, 2007.

[18] Shawn Anthony Bohner. A graph traceability approach for software change impact analysis. 1996.

[19] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[20] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[21] Mike Chi. *LibDetector: Version Identification of Libraries in Android Applications*. Rochester Institute of Technology, 2016.

[22] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *ACM SIGPLAN Notices*, 49(6):349–360, 2014.

[23] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and

Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM, 2017.

[24] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things you may not know about android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.

[25] Yue Duan, Mu Zhang, Heng Yin, and Yuzhe Tang. Privacy-preserving offloading of mobile app to the public cloud. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[26] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[27] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1037–1049. ACM, 2017.

[28] Jaakko Korpi and Jussi Koskinen. Supporting impact analysis by program dependence graph based forward slicing. In *Advances and innovations in systems, computing sciences and software engineering*, pages 197–202. Springer, 2007.

[29] Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 2(1-2):83–97, 1955.

[30] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.

[31] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 403–414. IEEE, 2016.

[32] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 335–346. IEEE, 2017.

[33] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, 2018.

[34] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[35] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 2–13. IEEE, 2017.

[36] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 342–352. IEEE, 2016.

[37] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722. ACM, 2016.

[38] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016.

[39] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013.

[40] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 128–137. ACM, 2003.

[41] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. Acm, 2012.

[42] Xiaoxia Ren, Barbara G Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th international conference on Software engineering*, pages 664–665. ACM, 2005.

[43] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[44] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. Technical report,

Georgia Institute of Technology, 2010.

[45] Raul Santelices, Mary Jean Harrold, and Alessandro Orso. Precisely detecting runtime change interactions for evolving software. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 429–438. IEEE, 2010.

[46] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[47] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, volume 2012, 2012.

[48] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *ACM SIGPLAN Notices*, volume 42, pages 112–122. ACM, 2007.

[49] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 165–176. ACM, 2014.

[50] Haoyu Wang and Yao Guo. Understanding third-party libraries in mobile app analysis. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 515–516. IEEE, 2017.

[51] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 25–36. ACM, 2014.

[52] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[53] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[54] Mu Zhang and Heng Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, February 2014.

[55] Mu Zhang and Heng Yin. Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14)*, 2014.

[56] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 9–18. ACM, 2013.

[57] Xuewen Zhang, Yuanyuan Zhang, Juanru Li, Yikun Hu, Huayi Li, and Dawu Gu. Embroidery: Patching vulnerable binary code of fragmentized android devices. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 47–57. IEEE, 2017.

[58] Yiji Zhang and Raul Santelices. Prioritized static slicing for effective fault localization in the absence of runtime information. Technical report, Technical Report TR 2013-06, CSE, U. of Notre Dame, 2013.

[59] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE, 2018.

[60] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

Table 4: Effectiveness Results By Vulnerability Category

Vulnerabilities	Total	Failures	Failure Rate
Race Condition	3	0	0%
Improper Access Control	2	0	0%
Uncontrolled Resource Consumption	1	0	0%
System Hang	5	0	0%
Illegal Reflective Access	1	0	0%
Stack Overflow	7	0	0%
Heap Access Error	3	0	0%
Missing Initialization	2	0	0%
Integer Overflow	1	0	0%
Fake ID	1	0	0%
New Security Feature	1	0	0%
Memory Leak	38	1	2.63%
Uncaught Exception	28	2	7.14%
Data Handling Error	36	3	8.33%
Uncheck Return Value	7	1	14.28%
Improper Input Validation	34	5	14.7%
Info Leak	3	1	33.33%

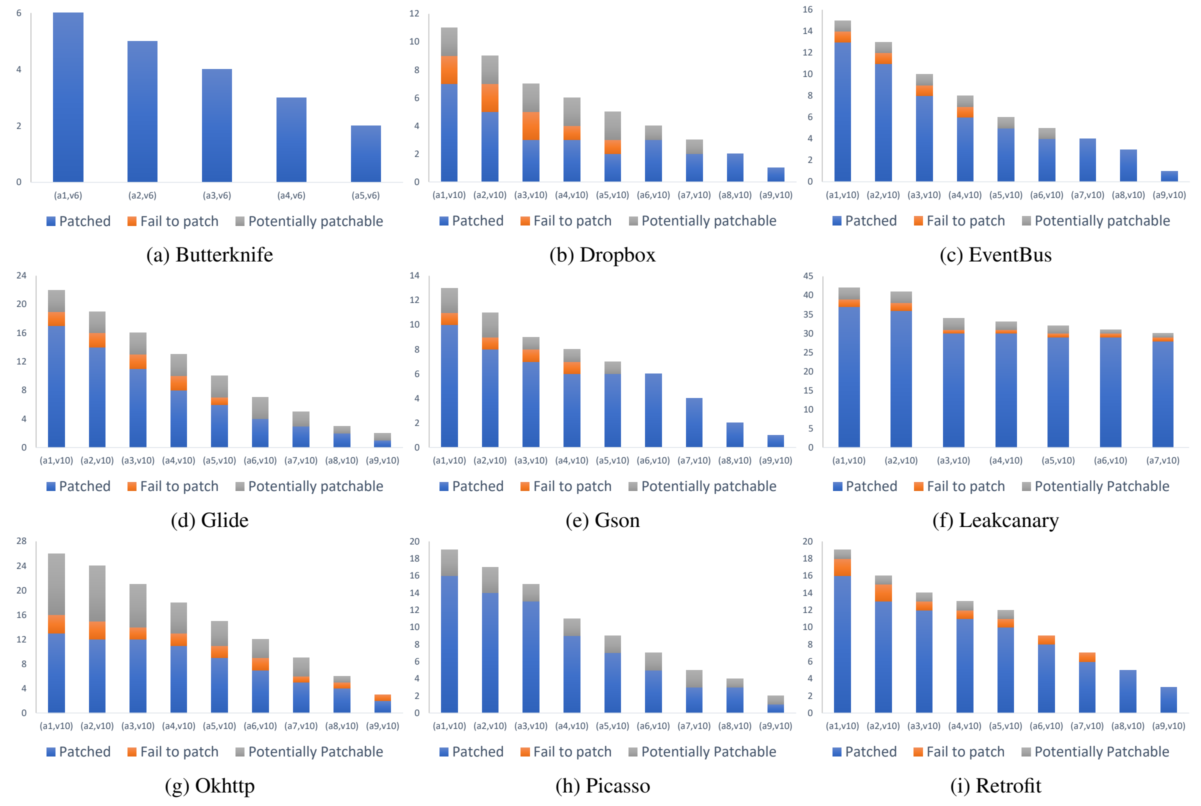


Figure 4: Effectiveness Results By Numbers

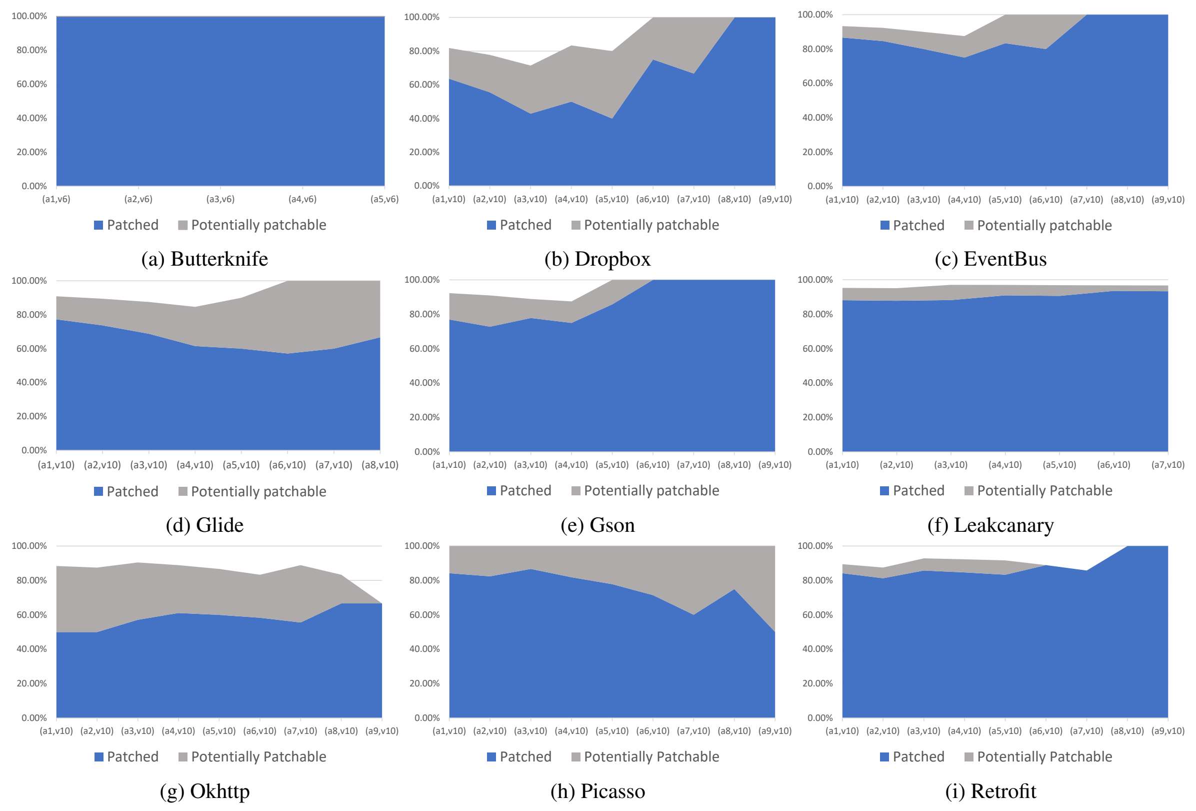


Figure 5: Effectiveness Results by Percentage

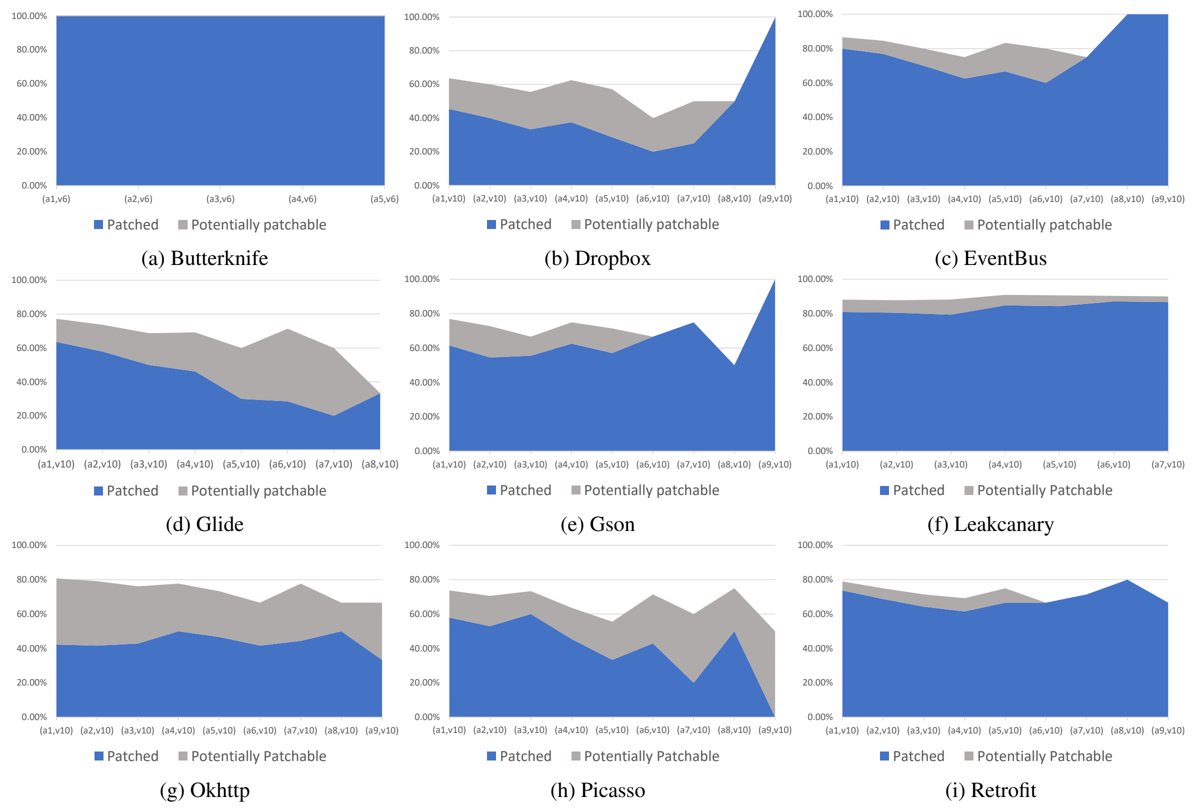


Figure 6: Effectiveness Results with Traditional Slicing