2023 IEEE International Conference on Data Mining Workshops (ICDMW)
DOI: 10.1109/ICDMW60847.2023.00172

# SeqAdver: Automatic Payload Construction and Injection in Sequence-based Android Adversarial Attack

1st Fei Zhang
*College of Intelligence and Computing*
*Tianjin University*
Tianjin, China
zhangfei@tju.edu.cn

1st(co) Ruitao FENG
*School of Computing and Information Systems*
*Singapore Management University*
Singapore
rtfeng@smu.edu.sg

2nd Xiaofei Xie
*School of Computing and Information Systems*
*Singapore Management University*
Singapore
xfxie@smu.edu.sg

3rd Xiaohong Li
*College of Intelligence and Computing*
*Tianjin University*
Tianjin, China
xiaohongli@tju.edu.cn

4th Lianshuan Shi
*School of Information Technology and Engineering*
*Tianjin University of Technology and Education*
Tianjin, China
shilianshuan@sina.com

***Abstract*—Machine learning has achieved a great success in the field of Android malware detection. In order to avoid being caught by these ML-based Android malware detection, malware authors are inclined to initiate adversarial sample attacks by tampering with mobile applications. Although machine learning has high capability, it lacks robustness against adversarial attacks. Currently, many of the adversarial attacking tools not only inject dead code into target applications, which can never be executed, but also require the injection of many benign features into a malicious APK. This can be easily noticeable by program analysis techniques. In this paper, we propose SeqAdver, an automatic payload construction and injection tool, which aims to bring the adversarial attack to the next level by injecting a payload that allows execution without breaking the app's original functionalities. These payloads are obtained from benign APKs at the Smali level and normalized into usable code snippets. The extracted Smali codes are carefully selected by filtering out 'user-visible' APIs and Intents. Therefore, payloads are able to be executed without any visible change noticed by the user. Besides, extracted payloads can be injected into different locations of the file based on sequence position or on the launcher class. Experiments were conducted to prove that randomly extracted payloads from benign apps are able to execute without causing any 'user-visible' behaviors or crashing the app when running the app in Android emulators.**

***Index Terms*—Android malware, Adversarial attack, Payload injection**

## I. Introduction

The past few years have witnessed a huge increase in mobile apps providing various facilities for personal and business use. As the most popular operating system on mobile devices, Android has more than 2.5 billion active users spanning over 190 countries [1]. However, various apps brought not only convenience but also security threats through Android malware. So, the classification of Android malware is becoming more and more important. Various learning-based works on Android malware detection [2]–[7] have achieved great success. Correspondingly, adversarial attacks [8] against ML-based detection have also become popular.

Feature space perturbation [9] is one of the most common adversarial attacks, where features of a certain class (e.g., class A) are randomly inserted into features from the other class (e.g., class B), such that it causes a misclassification (i.e., B→A). Studies [10] have shown that even state-of-the-art classifiers such as DREBIN [11], can be susceptible to such attacks. In addition, studies [10] have also shown an exponential relationship between the number of features inserted and the size of the repackaged app. However, the methods mentioned above have several disadvantages.

Firstly, a larger app with many newly added features will slow down the app and may lead to the loss of users. In addition to larger apps, there were no considerations of the features invisible to the user. Another disadvantage is the injection of unreachable code, which is also known as dead code. Current advanced program analysis techniques can easily detect dead code, even though using the Opaque Predicates Generation technique to hide them [12]. This causes learning-based detectors to ignore them and result in an unsuccessful adversarial attack. Lastly, the study above randomly selected features to inject, which may result in unusual behavior visible to the user, such as a crash of the app. Therefore, it is important to create a more optimized tool for perturbation where a lesser number of features are required to be injected for successful misclassification. We also need to consider selecting features with behaviors invisible to the user as well as ensure no dead code injection.

Considering all the above limitations, this paper creates a tool that manipulates Android APKs with effective payloads that bypass ML-based Android malware detectors. The tool extracts code from benign APKs based on the effective subsequences leading to misclassification. The tool can modify the extracted code into a usable payload that can execute in the targeted APK without causing any side effects such as a crash of the repackaged app. A list of features that is not usable will also be created. These features in the list are the features that may result in unwanted side effects that disrupt the app's original functionalities. Verification shall be done to test the usability of the repackaged app by the tool.

In summary, this paper makes the following contributions:

- Providing an automated adversarial sample generation tool, SeqAdver, which uses the features based on 'user-invisible' Android APIs and Intents to bypass traditional program analysis techniques.
- A method to extract Smali snippets from APKs especially from benign APKs for adversarial attacks. Windows size of subsequence in benign APK is also analyzed using an explanatory method to determine the most effective subsequence for misclassification. Thereafter, Smali methods which contain the desired subsequence are extracted before normalized into usable payload for adversarial attacks.
- A method to craft a usable payload that is able to execute without crashing the app despite injecting into any location. Normalization of payload with five factors were considered before crafting a usable payload. ACVTool [13] and Monkey [14] were used to verify the execution of the payload.
- Generated lists of visible APIs and Intents allow others to filter away 'user-visible' Android APIs and Intents, which cause user visible behaviors that make the user suspicious. These lists were generated through the extraction of the Android developers' documentation.

## II. Background and Preliminaries

This section briefly introduces the background of Android apps, the usage of Smali files, some examples of adversarial attacks, and the considerations required during a repackaging process.

### A. Android APK

Android applications are presented in the form of Android Application Packages (APKs) before installing on Android mobile phones. APKs are in the form of a zip file that contains many files, such as the binary AndroidManifest XML file, classes DEX files, resources, assets, and certificates. APKs can easily be obtained from the third-party websites such as APKmirror [15], APKpure [16], and Aptoide [17].

The AndroidManifest file (i.e., AndroidManifest.xml) involves all the metadata of the APK, such as the Android API level, the package name of the application, the permission required, Intent filters, services and activities in the app. Besides, when an app starts up, the information about which activity was first launched is also included in the AndroidManifest file.

Classes DEX files (i.e., classes.dex) are Dalvik executable code that can be executed in Dalvik virtual machine (DVM). Classes DEX files contain Dalvik bytecode that consists of class information, the data, and instruction sets for execution. The classes DEX file can be disassembled into Smali files which will be covered in more detail in section II-B. An APK may contain more than one class's DEX file. However, a single class DEX file is more commonly seen.

### B. Smali files

Smali files are files obtained when disassembling the classes DEX file using disassembler tools such as baksmali [18]. Smali files consist of human-readable assembly instructions, class information, and data in each class. Each Smali file has a certain structure that involves the class name, the super class's name, the interfaces that the class implements, the annotations, the instance fields, the direct methods, and the virtual methods. We will mainly be focusing on the class name, the direct methods, and the virtual methods.

The class names are always in the full file path which has a forward slash, '/', as a separator between each directory. All class names have to begin with a capital letter '*L*' and end with a semicolon '*;*'. An example of the class name is "*Lcom/socialnmobile/colornote/activity/NormalActivity;*".

The start of the direct methods section and virtual methods section can be seen after the line "*#direct methods*" and "*#virtual methods*" respectively. Direct methods consist of the class's constructor and private access modifier methods. Virtual methods consist of event methods such as *onCreate()*, as well as protected and public access modifier methods. They are both sorted in ascending order based on the method names in the direct or virtual methods section.

In Smali files, there are a number of registers ranging from *v0* to *v15*. Each register begins with a small letter '*v*'. Besides the normal registers, there is also the parameter register which starts with the small letter '*p*'. The parameter *p0* always consists of the current instance of the current class. This is often accessible by the "*this*" keyword in Object Oriented Programming (OOP) languages such as Java.

The parameters of each method in Smali are very different from the syntax of many high-level programming languages. For non-primitive types, the full class name beginning with a capital letter '*L*' and ending with a semicolon '*;*' is indicated. An example is "*Landroid/app/Activity;*". For primitive types such as integer, float, etc, they are represented in a single letter. The representation for primitive types can be seen in Table I.

TABLE I
PRIMITIVE TYPES REPRESENTATION IN SMALI

| Primitive types | Smali representation |
|---|---|
| Void | V |
| Byte | B |
| Short | S |
| Char | C |
| Integer | I |
| Long | J |
| Float | F |
| Double | D |
| Boolean | Z |

### C. Adversarial attack

An adversarial attack is a type of attack that specifically aims at learning-based detectors. One of the most common adversarial attacks is based on feature addition, where features of a certain classification group (e.g., class A) are randomly inserted into features from the other classification group (e.g., class B), such that it causes a misclassification (i.e., B→A). Alternatively, an adversarial attack can also be based on feature elimination where we remove features that define a class (e.g., class B). The remaining features are the ones usually determined as another class (e.g., class A). As a result, misclassification (i.e., B→A) also occurs.

### D. Repackaging process

Repackaging of APKs requires a number of steps. Firstly, as a compressed file, an APK file has to be unzipped into a folder. This can be easily accomplished by using 7Zip [19] or the ZipInputStream [20] library in Java programming. After unzipping the APK, baksmali [18] can be used to disassemble the classes.dex file into Smali files. Modifications to the Smali files can be made such as injecting a payload before assembling back to classes.dex using smali [18]. Similarly, injecting required permissions to the binary AndroidManifest file is also feasible by using xml2axml [21]. Next, the folder that contains all the APK's content can be zipped back using 7Zip [19] recursively into the .zip extension. The .zip extension should then be renamed into the .apk extension. If modifications were made to any of the files in the APK, it would be required to be signed using tools such as Jarsigner [22] or Apksigner [23]. A key might be required to be created using Keytool [24] that will be used by jarsigner [22] or Apksigner [23] to sign the APK. Lastly, Zipalign [25] is required to optimize the files in the signed APK.

### E. Interpretable method

The interpretable methods help researchers understand how a ML-based model works and which features play a more important role in the decision-making process of the model.In a ML-based detector, decisions are made if an APK is benign or malicious. By treating the detector as a black box, the interpretable method can be used to determine what causes the learning-based detector to make a detection. These interpretable methods can be applied to different models such as Decision Trees [26], Model Agnostic [27], and K Nearest Neighbours [28]. With the use of the interpretable method, an effective feature can be chosen to use as a payload for adversarial attacks on learning-based Android malware detectors.

## III. APPROACH

In this section, the overview of the SeqAdver (Fig. 1) will be introduced, as well as considerations when creating SeqAdver.

### A. Overview

SeqAdver is a tool that was created to automate the process of repackaging Android APK to conduct adversarial attacks. SeqAdver is written in Java which supports platform-independent and high modularity. As seen in Fig. 1, SeqAdver can be divided into two main subsystems which are (A) Payload Preparation and (B) Malware Manipulation. In (A) Payload Preparation subsystem, there are more subsystems such as (A1) Adversarial Sequence Analysis, (A2) Locating Code, and (A3) Normalization. In (B) Malware Manipulation, there are also a few subsystems which are (B1) Injection Location Selection, (B2) Payload Injection, (B3) Recompiling and Repackaging, and (B4) Validation. These subsystems will be discussed in more detail in the next few sections.

### B. Payload Preparation

During the preparation of the payload, a pre-trained model must be required (in step ①) as well as a dataset of benign applications (in step ②) to conduct Adversarial Sequence Analysis (in step A1). Different weights of different sequence elements will be produced using Local Interpretable Model-agnostic Explanations (LIME) [27]. The sequence elements with higher weight play a more important role in making the app to be benign by a ML-based malware detector. A window size of 3 sequence elements will be used to find the highest weight subsequence. A larger window size does not result in higher weight after experiments were tested using LIME [27]. Moreover, a higher window size of subsequence may cause a lower possibility of finding suitable Smali code to be used as they may face issues such as cross classes which will be further explained in section III-B2. Therefore, the most effective subsequence with a window size of 3 sequence elements will be generated and used in step ④.

*1) Locating Code:* In order to extract the Smali code from the Smali files that were obtained from the benign APK (in step ③), we must first obtain the file's order to know which Smali files we have to extract its content. This will allow us to know which classes (e.g., class b) contain the subsequence elements (e.g., 4th, 5th, and 6th elements of a sequence) in order. To know the order of the files to extract their Smali code, we can utilize Dexdump [29] to extract information from the code section from the classes.dex file. We can then filter the output by obtaining the class order starting from the top.

After obtaining the order of the classes, we need to verify which files contain the class we require. Despite the class's name (e.g., *Lcom/a/activity/b*) containing the directory's full path to it, there are some exceptions. For example, there is a

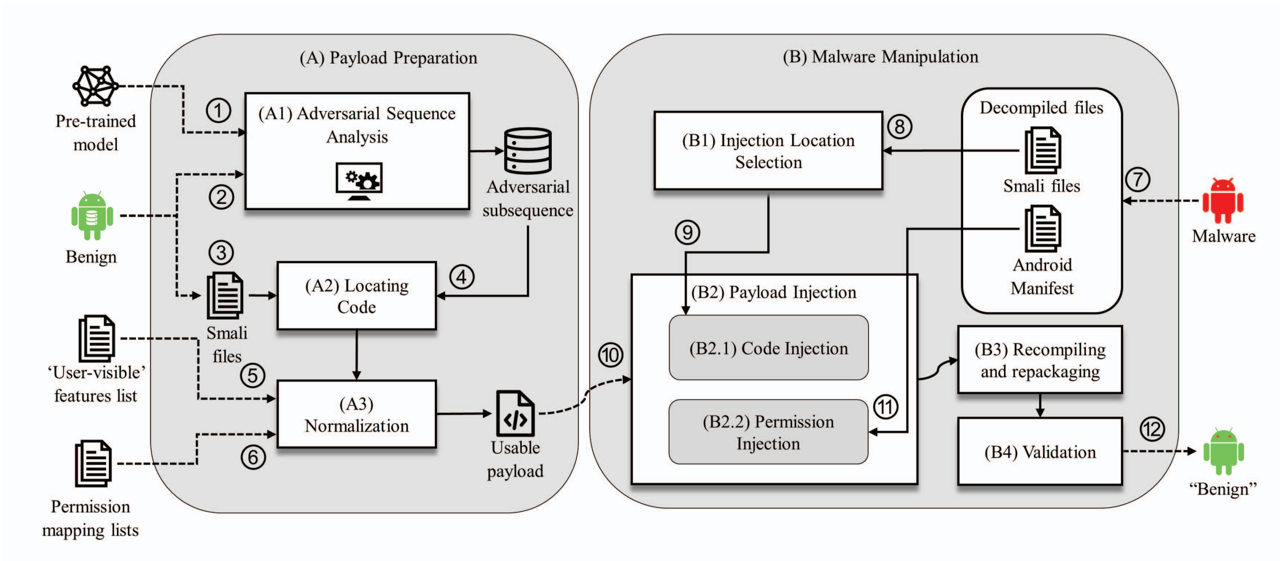


Fig. 1. Overview of SeqAdver

misconception that class A's file name will be *A.smali* while class A's file name is a.smali. However, the actual file name for class A is *a.smali* while class A's file name is *a.1.smali*. Therefore, it is important for us to verify if the supposed file contains the class we are searching for. As a result, it is required to search for the supposed file, check the class in it, and search for other files in the same directory and subdirectory until the file that contains the class is found.

After obtaining the files in order, we are then able to extract the information of the class in the file such as the class name, the file location of this class, the Smali code of each method in the class, the sequence's elements in each method if there is, whether the sequence's element is 'user-visible', and the position of the sequence's element in the sequence list of the benign app.

Based on the list of entity objects that consist of all the Smali information, the Smali method(s) that contains the Adversarial Subsequence (in step 4) is extracted and will be normalized in step A3.

*2) Normalization:* In order to make sure the extracted Smali code is useful, it must be normalized into a payload in step A3 (see Algorithm 1). During the normalization process, we need to consider many aspects, such as removing the comments, the considerations when the payload only consider one method, subsequence spreading across multiple class with no method call to the next, and finally the extracted method are event methods (e.g., *onCreate()*).

Regardless of different situations to handle when normalizing the code, they have the same fundamentals which are to remove comments and unwanted lines of Smali code. The removal of unwanted lines of Smali code can help to reduce the dependencies needed. For example, there could be a line of Smali code that invokes a method call to another user-defined method in the same class or another class. This causes more issues as we will need to normalize those methods being called. Therefore, normalization of the payload involves removing as much unnecessary code as possible. To determine if a line of code is required, we are able to use the backward traversal technique to determine if any of our subsequences requires those lines of Smali code (see Algorithm 2).

```
1  .method public a()V
         ...
6        invoke-virtual {v1}, Lcom/a/c/q;->h()Z
7        move-result v2
8        invoke-virtual v2,
         Landroid/app/Activity;
         ->setActionBar(Landroid/widget/Toolbar;)V
9        return-void
10 .end method
```

Listing 1. Example of the backward traversal technique

The backward traversal technique (see Algorithm 2) heavily relies on the registers used by our subsequence. Based on the registers used, the registers will be saved in a set of registers (i.e., set $R$ in Algorithm 1). An algorithm is used to traverse upwards and determine where each register in set $R$ (e.g $R = \{v0, v3, v4\}$) obtains its value from. This is known as register dependencies. If any of the lines of code writes to any of the registers in set $R$, that line of Smali code will be saved. If that line of Smali code requires any other registers, those registers will be added to set $R$. This example is shown on Listing 1 where *setActionBar()* is part of our subsequence. Register *v2* is added to set R and backward traversal occurs. Once we discover that *h()* writes to *v2* in line 7, we will save both lines 6 and 7. Register *v1* from line 6 will be added to set $R$ while *v2* is removed. Backward traversal will continue to traverse upwards till the start of the method. In line 17 of Algorithm 2, separated Table II is created as the algorithm to parse the strings to obtain the registers are different. The rest of the algorithms for these two tables are the same, starting from line 18 in Algorithm 2.

**Algorithm 1:** Normalize extracted Smali methods

**Input:** List of line number to check for dependency $LD$; Smali code in a method $SC$; Class name $CN$
**Output:** Hashmap that may contain normalized payload and dependency file/class to be injected $HM$

$DR \longleftarrow ()$ `// Set of non-removed Smali lines`
$BR \longleftarrow ()$ `// Set of removed Smali lines`
$R \longleftarrow ()$ `// Set to store registers for backwards tracking technique`
$DC \longleftarrow ()$ `// Set of dependency class to be injected later`
$RT \longleftarrow ()$ `// Hashmap of result to be returned as output`
Append $LD$ to $DR$;
**for** $L \in LD$ **do**
  **if** *" invoke-"* $\in SM_L$ **then**
    **if** *" move"* $\in SM_{L+2}$ **then**
      **if** $SM_{L+2}$ *consist of event parameters* **then**
        Add ($L$ + 2) to $BR$; `// Remove line that writes result to parameter register`
      **else**
        Add ($L$ + 2) to $DR$; `// Safe to save this line of Smali code`
    Append dependency register in $SM_L$ to $R$;
  $SC \longleftarrow$ Result from Algorithm 2;
**for** $i = 0$ *to* ($SC_s$ *- 1)* **do**
  **if** *" invoke-"* $\in SC_i$ *and " invoke-"* $\notin CN$ **then**
    Append class of method being invoked to $DC$;
Add $SC$ to $RT$;
Add $DC$ to $RT$;
**return** $RT$;

There will be situations where our extracted codes are event methods during normalization. As a result, we will first be required to rename these methods into a random string (line 8 in Algorithm 3). Retaining the method names may result in errors when injecting into the targeted APK as there may be another event method with the same name as the methods in our payload. If there is more than one event method as our payload, we will need to inject method calls in subsequent order (from lines 10 to 13 in Algorithm 3). Listing 2 shows an example of injecting the invoke method B statement into the end of method A just before the method RETURN statement on line 11. Injecting of invoke statements also occurs in handling normal methods that do not have method calls in

**Algorithm 2:** Backward traversal (during normalization)

**Input:** List of line number to check for dependency $LD$; Smali code in a method $SC$; Set of lines of Smali code not to be removed $DR$; Set of Smali lines to be removed $BR$; Set of dependency registers during backwards traversal $R$; Current line number to check for dependency $L$
**Output:** Normalized Smali code $SC$

1 **for** $i$ = *(L - 1) to 0* **do**
2   **if** *" invoke-"* $\in SC_i$ *or " filled-new-array"* $\in SC_i$ **then**
3     **if** *" move-result"* $\in SC_{i+2}$ **then**
4       **if** *(register* $\in SC_{i+2}$*)* $\in R$ **then**
5         Remove register in $R$; `// Remove found dependency register`
6         Append ($i$, $i$ + 2) to $DR$; `// Add line numbers to be saved`
7         Remove ($i$, $i$ + 2 from $BR$; `// In-case wrong line added`
8         Append registers in $SC_i$ to $R$; `// Add new dependency registers`
9       **else if** *(i, i + 2)* $\notin DR$ **then**
10         Append ($i$, $i$ + 2) to $BR$;
11     **else if** *";-><init>("* $\in SC_i$ **then**
12       **if** *(register* $\in SC_i$*)* $\in R$ **then**
13         Append $i$ to $BR$;
14         Remove $i$ from $DR$;
15       **else if** $i \notin DR$ **then**
16         Append $i$ to $BR$;
17   **else if** *Any string in Table II* $\in SC_i$ **then**
18     **if** *(register that is being written RW* $\in SC_i$*)* $\in R$ **then**
19       Remove $RW$ from $R$;
20       Append new dependency register to $R$;
21       Append $i$ to $DR$;
22       Remove $i$ from $BR$;
23     **else if** $i \notin DR$ **then**
24       Append $i$ to $BR$;
25   **else if** *" iget-"* $\in SC_i$ *or " sget-"* $\in SC_i$ **then**
26     **if** *(register* $\in SC_i$*)* $\in R$ **then**
27       **return** *NULL;* `// Reject code depend on class's field`
28     **else**
29       Append $i$ to $BR$;
30   **else if** *" iput-"* $\in SC_i$ *or " sput-"* $\in SC_i$ **then**
31     Append $i$ to $BR$;
32   **if** *Any register* $\in R$ *is parameter register* **then**
33     **return** *NULL;* `// Reject code if parameter register used`

```
for i ∈ BR do
    SC_i = " nop"; // Change to NOP instruction instead of deleting
M ⟵ Maximum value in DR;
N ⟵ Minimum value in DR;
MN ⟵ Method name from SC_0
Remove Smali lines from SC_{M+1} to SC_s;
Remove Smali code from SC_0 to SC_N;
Prepend ".method public " + M + "()V" SC_0;
Append " " to SC_s;
Append " return-void" to SC_s;
Append ".end method" to SC_s;
return SC;
```

TABLE II
LIST OF STRINGS FOR COMPARISON

| | | | | |
|---|---|---|---|---|
| const- | const/ | const | check-cast | array-lengt |
| aget | instance-of | new-instance | new-array | filled-array-data |
| move | move/ | neg- | int- | long- |
| float- | double- | add- | sub- | rsub- |
| mul- | div- | rem- | and- | or- |
| xor- | shl- | shr- | ushr- | |

subsequent order.

During normalization, if there are any dependency methods (methods in the Smali code that are not allowed to be deleted) that have 'user-visible' functionalities (in step ⑤), the extracted code will be rejected and step ④ will be repeated. When repeating step ④, SeqAdver will search for the subsequence in another area of the benign APK before extracting the methods that have the same subsequence.

```
.method public A()V
        ...
    invoke-virtual , Lcom/test;->B()V
    return-void
.end method

.method public B()V
        ...
```

Listing 2. Injecting invoke method B statement into method A

Besides rejecting 'user-visible' features, extracted codes are also rejected if they contain dependency lines of Smali code (lines of code that cannot be deleted) that use the parameter registers or use any values from the fields of the class. This is due to the fact that it is difficult to know what values will be supplied to the parameters as well as to the fields of the class.

### C. Malware Manipulation

The Malware Manipulation (in step B) focus on modifying the malware APK for adversarial attack. In this subsystem, a malware APK will be decompiled to obtain Smali files from classes.dex and also obtained the binary Android Manifest file after unzipping the APK (in step ⑦). In step ⑧, the user can decide to inject the payload in the launcher class or any

**Algorithm 3:** Normalize event APIs

```
Input: Class Information instance CI; List of
       subsequence S; List of event method names
       EN
Output: Usable payload UP
LD ⟵ [] // Line numbers to check for dependency
SM ⟵ Get list of Smali methods from CI based on S;
for i = 0 to (SM_s - 1) do
    S ⟵ SM_i; // Get Smali codes of each Smali method
    for j = 0 ∈ S_i do
        if ".method" ∈ S_j then
            if Any EN ∈ S_j then
                Generate a new method name and replace the method name in S_j;
            OS ⟵ SM_i-1;
            if previous method in CI(i - 1) do not have call to this method then
                Insert invoke call to SM_{i-3}; // Insert before return statement
                Append (i + 3) to LD;
                Insert " " at SM_{i-3}; // Empty line between invoke and return statement
            else
                for (k = OS_s - 1) to 0 do
                    if OS_k contains invoke call to SM_i then
                        Remove invoke call arguments;
                        Append k to LD;
Append S to LD;
UP ⟵ Result of Algorithm 1;
return UP;
```

location based on the sequence position of the malware APK. Permissions are then added to the binary Android Manifest file based on the payload (in step ⑪). The modified APK will then be recompiled and repackaged in step B3 before validation is conducted in step B4 which will be explained in detail in section IV. After validation, we can then conclude if the adversarial attack is successful in step ⑫.

*1) Payload injection:* There are currently two methods to inject a payload in SeqAdver. The first method is to inject the payload into the launcher class. The second method is to inject into the class that consists of the sequence position we would like to inject our payload into.

*a) Injection of payload in the launcher class:* This is a very commonly used method especially in injecting malicious payload into benign APK. This is because it ensure the execution of the payload. When the app first started, the launcher class which is an Activity class will definitely be executed. As a result, we can inject our payload into the launcher class and add an invoke statement in our payload in the *onCreate()* method. During the start-up of Activity, the *onCreate()* is definitely executed. Therefore, putting the payload in the launcher class and adding the payload into the *onCreate()* method will guarantee the execution of our payload. This is especially useful when we would like to test if our payload is able to execute successfully without crashing the app. Experiments and results have shown that our crafted payload was able to execute without crashing the app during code instrumentation in section IV. We can obtain the launcher class from the binary AndroidManifest file where the Activity will consist of an intent filter with *<action>* and *<category>* tags that have "*android.intent.action.MAIN*" and "*android.intent.category.LAUNCHER*" in their attributes respectively. This example can be seen in Listing 3, which shows that the launcher class is "*com.appyet.activity.SplashActivity*" (font in red) as it has *<intent-filter>* tag with "*android.intent.action.MAIN*" and "*android.intent.category.LAUNCHER*" (font in blue) in the *<action>* and *<category>* tags respectively.

```
<activity android:label="@string/app_name" android:
    launchMode="singleTask" android:name="
    com.appyet.activity.SplashActivity" android:
    theme="@android:style/Theme.NoTitleBar">
            <intent-filter>
                <action android:name=
                "android.intent.action.MAIN"/>
                <category android:name=
                "android.intent.category.LAUNCHER"/>
            </intent-filter>
</activity>
```

Listing 3. Example of launcher activity in AndroidManifest.xml

*b) Injection based on the requested position of the APK's sequence:* The injection position can be specified by an external module using SeqAdver. The injection position is usually injected in the most optimal position where that is a higher rate of misclassification if injected in a certain area. To inject into specified position, we need to rename our payload methods as the methods are sorted in lexicographically ascending order according to method names. To rename our payload method names, the name of the method (e.g., *method public az()*) will be used and appended with the letter 'a' (e.g., *method public aza()*). If there is more than one method in our payload, the letter will increment by one (e.g., 'a'$\rightarrow$'b') for each payload method. It is also important to update the method name in the subsequent method call to the next payload's method. These can be seen in Listing 4 where the blue font is the payload methods injected after method *az()*.

Besides changing the method name, we also need to modify the accessor modifier to private or public, depending on the method we would like to inject our payload into, is in the direct or virtual methods sections (see the orange font in Listing 4). After modifying the accessor modifier to private or public, we need to change the invoking statement of the payload methods to direct or virtual. An invoke statement will also be added to the end of *onCreate()* or the constructor (i.e., *<init>()*) to invoke the start of the payload. This can be seen in the red font in Listing 4.

*2) Permission injection:* After injecting the payload into the targeted APK (step B2.1), it is important to inject the required permission into the AndroidManifest file in the targeted APK (in step B2.2). After injecting the payload into the targeted APK (step B2.1), it is important to inject the required permission into the AndroidManifest file in the targeted APK (in step B2.2). However, there are no official tools from Google that allow us to obtain the permission required based on APIs and Intents. There were a number of tools such as PScout [30] and Arcade [31] which were no longer maintained or available. There are also a number of newer tools but the tools are not available for others to use as only the results are available after running on the Android Open Source Project (AOSP). These tools are Axplorer [32] [33] and Arcade [31] [34]. There are also results of permission mapping from PScout and PScout II [35] available on Github sources. As a result, the permission mappings of multiple sources are combined into a few lists categorizing them based on the API level, followed by API and Intents. There are some limitations as most of the results available online only have permission mapping between API 8 to 19 and 21 to 25. Therefore, those APKs below API 8 will be using API 8's permission mapping, APKs with API 20 will be using API 19, and those APKs above API 25 will be using API 25's permission mapping. Table III shows a summary of the API level permission being mapped based on the API level of the APK, the API, and the Intent.

```
.method public constructor <init>()V
          ...
    invoke-direct , Lcom/a/b;->aza()V
    return-void
.end method

.method private az()
          ...
.end method

.method private aza()
          ...
       invoke-direct , Lcom/a/b;->azb()V
       return-void
.end method

.method private azb()
          ...
.end method
```

Listing 4. Injecting payload methods into direct methods section

TABLE III
PERMISSION MAPPING FROM API 1 TO 30

| API Level of APK | API | Intent |
|---|---|---|
| 1-8 | API 8 | API 8 |
| 9-18 | API 9-18 | API 9-18 |
| 19-20 | API 19 | API 19 |
| 21-24 | API 21-24 | API 21-24 |
| 25-30 | API 25 | API 25 |

```
.method public c()V
        .locals 3

        new-instance v1, Ljava/lang/StringBuilder;
        invoke-direct {v1}, Ljava/lang/StringBuilder;->()V
        const-string v2, "Invalid widgetType:"
        invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
        return-void
.end method
.method protected onCreate(Landroid/os/Bundle;)V
        .locals 2

        invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
        const v0, 0x7f03005c
        invoke-virtual {p0, v0}, Lcom/appyet/activity/SplashActivity;->setContentView(I)V
        invoke-virtual {p0}, Lcom/appyet/activity/SplashActivity;->getApplicationContext()Landroid/content/Context;
        move-result-object v0
        check-cast v0, Lcom/appyet/context/ApplicationContext;
        iput-object v0, p0, Lcom/appyet/activity/SplashActivity;->b:Lcom/appyet/context/ApplicationContext;
        const v0, 0x7f070096
        invoke-virtual {p0, v0}, Lcom/appyet/activity/SplashActivity;->findViewById(I)Landroid/view/View;
        move-result-object v0
        check-cast v0, Landroid/widget/LinearLayout;
        iput-object v0, p0, Lcom/appyet/activity/SplashActivity;->a:Landroid/widget/LinearLayout;
        iget-object v0, p0, Lcom/appyet/activity/SplashActivity;->a:Landroid/widget/LinearLayout;
        iget-object v1, p0, Lcom/appyet/activity/SplashActivity;->b:Lcom/appyet/context/ApplicationContext;
        iget-object v1, v1, Lcom/appyet/context/ApplicationContext;->q:Lcom/appyet/metadata/Metadata;
        iget-object v1, v1, Lcom/appyet/metadata/Metadata;->MetadataSetting:Lcom/appyet/metadata/MetadataSetting;
        iget-object v1, v1, Lcom/appyet/metadata/MetadataSetting;->SplashScreenBgColor:Ljava/lang/String;
        invoke-static {v1}, Landroid/graphics/Color;->parseColor(Ljava/lang/String;)I
        move-result v1
        invoke-virtual {v0, v1}, Landroid/widget/LinearLayout;->setBackgroundColor(I)V
        invoke-virtual {p0}, Lcom/appyet/activity/SplashActivity;->c()V
        return-void
.end method
```

Fig. 2. ACVTool report of successful injection and execution of the payload

When inserting permissions into the binary AndroidManifest using Xml2axml [21], it is important to take note of the current permissions that already exist in the targeted APK. It is not ideal to insert permission that already exists. Having duplicated permission(s) might cause detectors to suspect that the app has been repackaged [36].

*3) Recompiling and repackaging:* After modifications are done, we will have to zip the folder back into an APK file (.apk extension) in step ⑧ from Fig. 1. Zipping of the folder must be done recursively. If the zipping process is not done recursively, there will be errors when executing the app on the mobile phone where it will not be able to retrieve the resources in the APK. This can easily be checked by unpacking APK files using tools such as APKtool [37]. This will be further explained in section IV. Luckily, there are a number of tools that make zipping recursive easier such as 7Zip [19] and the zip command-line tool [38] in GNU/Linux.

After zipping the file, the APK file must be signed. This can be done using Jarsigner [22] or APKsigner [23]. SeqAdver uses Keytool [24] to create a key before using the key to sign the APK with Jarsigner [22].

Finally, the signed APK file must be optimized through archive alignment. During this process, the uncompressed data will be relatively aligned to the start of the file in the 4-byte boundary. This process can be done using Zipalign [25].

## IV. Experiments and Results

In this section, the goal of our experiment is to: (1) Verification the usability of the 'user-invisible' list; and (2) Evaluation on the injection success rate; Based on verification and results, we further evaluate the testing on different API version mobile emulation to test the success rate.

### A. Verification the usability of the 'user-invisible' list

The lists available in SeqAdver only consist of 'user-visible' lists. As a result, experiments are conducted with different variety of sequence elements to test if the repackaged application causes changes visible to the user such as a sudden change in the Graphical User Interface (GUI). Besides verifying the change in GUI, experiments were also conducted to test if the application crashes when the payload executes. If the execution of the experiment results in the results mentioned above, this means that the lists of 'user-visible' sequence elements are incorrect as not all 'user-visible' sequence elements are included in it. As a result, tests were done using Monkey [14] and manual testing.

Before conducting the experiment, a few conditions were set. Firstly the payloads were injected at the launcher class and invoked by *onCreate()* right before the *RETURN* statement is called. This allows quicker manual testing to validate if the application crashes. Next, different extracted code from the different benign APKs was extracted to inject into the same targeted APK. This allows a variety of extracted code with different possible subsequences used as payload and consistent results where the targeted APK should not have a change in behavior.

After manual and the use of Monkey [14], 89.47% of the applications tested did not crash and were able to execute without any issues. The remaining application crashes were mainly due to the normalization process where some parsing Smali lines of code were not considered but none of them were due to the sequence's elements in the payload. As a result, the lists of 'user-visible' sequence elements are usable for payload injections.

### B. Evaluation on the success rate of injection

The evaluation of the success rate involves testing of extracting different payloads from different benign APKs and injecting them into the different APKs. The end goal of this evaluation is to check if the payload will be successfully injected into the supposed injection location. After injecting a payload into a targeted location, compilation will be on all the Smali files in the targeted APK to form classes.dex using tools such as smali [18]. After compilation, the classes.dex can be decompiled again to check if the payload is still at the injected location. Alternatively, experiments were also tested using instrumentation techniques by using tools such as ACVTool [13] to test if the payload has been executed. If the payload is not successfully injected, the location that should contain the payload and the invoke statement to invoke the payload will not be injected. This can be noticed when decompiling the previously compiled classes.dex file. Alternatively, the payload will not be instrumented during the ACVTool [13] analysis phase and can be noticed in the report generated by ACVTool [13]. An example can be seen in Fig. 2.

### C. Verification across multiple API level phones

The further extend the previous experiments, the next experiment tested on different API levels of Android to test if SeqAdver is still able to inject 'user-invisible' payloads only where the payloads were executed and no crash occurred after the payload was executed. Five different API versions of Android emulators were tested in the experiment. They are Android 4.1 Jelly Bean (API 16), Android 4.4 KitKat (API 19), Android 7.0 Nougat (API 24), Android 9.0 Pie (API 28), and Android 11.0 R (API 30). In the experiment, all the repackaged APKs used in section IV.A were tested on these five versions of Android in emulators to test if the functionality was affected.

The results have shown that the repackaged APKs that were able to be executed in section IV-A, were also able to execute in all of these five versions of Android. As a result, the rate of success is still 89.47% as the repackaged APKs that crashed in section IV.A, crashes in other versions of Android as well. When testing Android 9.0 Pie (API 28) and Android 11.0 R (API 30), a permission prompt will appear when the application runs for the first time. This prompt will tell the user the application is trying to access those requested services, which may result in users noticing additional permissions required. Besides this prompt, the list of 'user-invisible' list is usable as there are no crashes or additional changes in GUI behavior when launching the app since the payload is injected at the launcher class.

## V. Limitations and Future Work

This section will introduce the limitations in the current version of SeqAdver and the future works that can be done to further improve SeqAdver.

### A. Limitations

*1) Subsequence cross classes:* During code location, areas of the code that contain subsequence are extracted to be used as a payload. However, there are situations where the subsequence spreads across more than one class. Currently, SeqAdver ignores areas of code that are spread across more than one class. This is due to the complexity of knowing what are the values of the parameters required for the other class's constructor and the method(s) to be called that continues our subsequence.

*2) Extracted code having parameter registers dependencies:* During code normalization, the backward traversal technique is used as stated in section III-B2. During backward traversal, there may be dependency lines of Smali code that use the parameter registers (e.g., register *p1*). The value to supply values to the parameters has too many possibilities which makes it NP-hard. However, there are possible solutions to this limitation which will be stated in section V-B1.

*3) Test 'user-invisible' adversarial samples on real Android phone:* The Android adversarial samples generated by SeqAdver were only tested on Android emulators and did not cause any changes in GUI. However, there are still some differences between Android real phones and emulators, which may lead to 'user-visible' behavior. This is because we need to ensure that the malicious samples can not damage the testing phone.

### B. Future work

*1) Extracted code having parameter registers dependencies:* An algorithm can be created to search for all methods in all files that call the payload method. The method (e.g *classZ::pb()*) that calls the payload method (e.g *classA::ab(Z)*) can also be included in the list extracted methods and normalization will be done to that method (e.g *classZ::pb()*). Using this technique, the need to guess the possible value needed to pass as an argument to call the payload method (e.g., *classA::ab(Z)*) will be eliminated.

*2) Train a robust detection model against SeqAdver:* With the help of SeqAdver, we can generate some adversarial samples that can bypass traditional analysis techniques. We will add these adversarial samples to the training set to train a more robust ML-based model in the future.

## VI. Conclusion

Our work builds a tool, SeqAdver, for the automatic generation of Android adversarial samples. SeqAdver provides a new technique by injecting executable payloads that are 'user-invisible' based on Android APIs and Intents instead of inserting dead code. As a result, traditional program analysis techniques will be bypassed leading to a successful adversarial attack. We speculate that it might be useful for other research on adversarial attacks.

## References


[1] (2021) Android statistics (2021). [Online]. Available: https://www.businessofapps.com/data/android-statistics/

[2] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[3] R. Feng, Y. Liu, and S. Lin, "A performance-sensitive malware detection system on mobile platform," in *Formal Methods and Software Engineering*, Y. Ait-Ameur and S. Qin, Eds. Cham: Springer International Publishing, 2019, pp. 493–497.

[4] R. Feng, S. Chen, X. Xie, L. Ma, G. Meng, Y. Liu, and S.-W. Lin, "MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform," in *2019 24th International Conference on Engineering of Complex Computer Systems*, 2019.

[5] R. Feng, J. Q. Lim, S. Chen, S.-W. Lin, and Y. Liu, "SeqMobile: An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices," in *2020 25th International Conference on Engineering of Complex Computer Systems*, 2020.

[6] R. Feng, S. Chen, X. Xie, G. Meng, S.-W. Lin, and Y. Liu, "A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices," *IEEE Transactions on Information Forensics and Security*, 2020.

[7] Q. Qiao, R. Feng, S. Chen, F. Zhang, and X. Li, "Multi-label classification for android malware based on active learning," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2022.

[8] H. Li, S. Zhou, W. Yuan, J. Li, and H. Leung, "Adversarial-example attacks toward android malware detection system," *IEEE Systems Journal*, vol. 14, no. 1, pp. 653–656, 2019.

[9] N. Inkawhich, W. Wen, H. H. Li, and Y. Chen, "Feature space perturbations yield more transferable adversarial examples," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7066–7074.

[10] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ml attacks in the problem space," in *IEEE Symposium on Security & Privacy (Oakland)*, 2020. [Online]. Available: https://arxiv.org/abs/1911.02142

[11] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," 02 2014.

[12] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, "A multi-study investigation into dead code," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 71–99, 2018.

[13] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[14] (2020) Ui/application exerciser monkey. [Online]. Available: https://developer.android.com/studio/test/monkey

[15] (2021) Apkmirror. [Online]. Available: https://www.apkmirror.com/

[16] (2021) Apkpure. [Online]. Available: https://apkpure.com/apk-install.html

[17] (2021) Aptoide. [Online]. Available: https://en.aptoide.com/

[18] (2021) smali/baksmali. [Online]. Available: https://github.com/JesusFreke/smali

[19] (2021) 7zip. [Online]. Available: https://www.7-zip.org/

[20] Zipinputstream. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/zip/ZipInputStream.html

[21] (2020) xml2axml. [Online]. Available: https://github.com/hzw1199/xml2axml

[22] (2020) Jarsigner. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html

[23] (2021) Apksigner. [Online]. Available: https://developer.android.com/studio/command-line/apksigner

[24] (2020) keytool - key and certificate management tool. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html

[25] (2021) zipalign. [Online]. Available: https://developer.android.com/studio/command-line/zipalign

[26] D. P. Kuttichira, S. Gupta, C. Li, S. Rana, and S. Venkatesh, "Explaining black-box models using interpretable surrogates," in *PRICAI 2019: Trends in Artificial Intelligence*, A. C. Nayak and A. Sharma, Eds. Cham: Springer International Publishing, 2019, pp. 3–15.

[27] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," 2016.

[28] E. Wallace, S. Feng, and J. Boyd-Graber, "Interpreting neural networks with nearest neighbors," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 136–144. [Online]. Available: https://www.aclweb.org/anthology/W18-5416

[29] (2018) Android sdk build tools 28. [Online]. Available: https://aur.archlinux.org/packages/android-sdk-build-tools-28/

[30] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 217–228. [Online]. Available: https://doi.org/10.1145/2382196.2382222

[31] Y. Aafer, G. Tao, J. Huang, X. Zhang, and N. Li, "Precise android api protection mapping derivation and reasoning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1151–1164. [Online]. Available: https://doi.org/10.1145/3243734.3243842

[32] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 1101–1118. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/backes_android

[33] (2017) Axplorer permission mapping result. [Online]. Available: https://github.com/reddr/axplorer/tree/master/permissions

[34] (2020) Arcade permission mapping result. [Online]. Available: https://github.com/arcade-android/arcade

[35] (2015) Pscout permission mapping result. [Online]. Available: https://github.com/zyrikby/PScout

[36] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[37] (2020) Apktool. A tool for reverse engineering Android apk files. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[38] (2008) zip(1) - linux man page. [Online]. Available: https://linux.die.net/man/1/zip