Malware Recomposition Attacks: Exploiting Feature Evolutions and Confusions in Android Apps

Abstract

Android malware have grown significantly in recent years. Existing work employs either signature-based or semantics-based techniques to defend against malware by detecting "common" malicious behaviors. However, most of (if not all) these defenses are always one-step behind malware developers since these defenses are derived from the analysis of previously *known* malware. Moreover, there exists no effective mechanism for defeating the evolution of malicious code from the same origin. To address this issue, in this paper, we present a novel approach named as *malware recomposition variation (MRV)* that conducts semantic analysis of existing malware to systematically reconstruct new malware variations. In particular, we target at two types of attacks: malware evolution attack and confusion attacks, with different variation strategies. Upon given malware, we conduct both semantic-feature mutation analysis and phylogenetic analysis to synthesize mutations, and then conduct program transplantation to automatically apply mutations on malware bytecode. In this way, our approach aims to produce malware variations that can have high likelihood to evade detection while still retaining their malicious behaviors. Our empirical evaluation on 3,852 Android benign apps and malware (originated from more than 35 families) shows that resulting malware variants can effectively evade (or mislead) detection techniques based on signatures or pattern learning. To counter against this type of attack, we provide insights on how to improve the current defense techniques.

1 Introduction

Along with the exponential growth on markets of mobile applications (*apps* in short), comes the frequent occurrence of mobile malware on app markets. From the McAfee Security Report 2016[1], there are more than 37 million mobile malware samples detected in the past six months. To fight against malware, a signature-based technique extracts malicious behaviors as signatures (such as bytecode or regular expression) while more complicated machine-learning-based techniques learn discriminant features from analyzing information flows and semantics of malware.

Table 1: Examples of feature values for malware detection

App	Ground-truth	f_1	f_2	f_3	Detection result
M_1	Malware	a	b	c	Malware
M_2	Malware	a	b	c'	Benign
B_1	Benign	a	b	c'	Benign
B_3	Malware	a'	b'	c	Malware
M_v	Malware	a'	b'	c'	?

Columns f_1 to f_3 are three feature columns.

A key observation made in our research is that, features, which abstract concrete malicious functionalities or behaviors, are fragile, and they could be easily mutated or changed. The susceptibility of such features makes it possible to evade detection if malware are properly mutated or changed [17, 45, 52]. Therefore, a question naturally follows: *can a malware be evolved/-mutated to evade the detection by changing its feature values while maintaining its malicious functionalities*[2]*?* More formally, we name such "mutations of malware based on feature values" as *Malware Recomposition Variation (MRV)*, and we focus on synthesizing mutation strategies (*i.e.,* what kind of features we should mutate to evade detection) and automating program transformation (*i.e.,* how to apply mutations on malware bytecode to ensure the robustness of the app while preserving malicious functionality). Different from existing work [17, 45, 52], we explore the capability of attackers in a more realistic attacking scenario: the attackers can feed any app as the input to the detector and know the binary detection result (*i.e.,* detected or not) without any additional information.

[1]`http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf`

[2]We define malicious functionalities/behaviors as the invocations of security-sensitive method calls in malware, more specifically in Android, the invocations of permission-protected methods.

An illustration. The goal of malware detection is to classify an app as "malware " or "benign". In MRV, we can change feature values V_i of a particular malicious app i, *e.g.,* changing feature values $\langle a, b, c \rangle$ of malicious app M_1 in Table 1. In order to evade detection, intuitively, we can mutate feature values V_i in three strategies: (i) *look-like-benign-app*: mutating V_i to be exactly the same as the feature values of a benign app, *e.g.,* $\langle a, b, c' \rangle$ of B_1 in Table 1; (ii) *look-like-misclassified-malware*: mutating V_i to be exactly the same as the feature values of a malware being misclassified as benign, *e.g.,* $\langle a, b, c' \rangle$ of M_2; (iii) *look-like-unclassifiable-app*: mutating V_i to the feature values of an app that the malware detector cannot draw any classification conclusion (either malware or benign), *e.g.,* $\langle a', b', c' \rangle$ of M_v.

However, there are three major challenges (C1–C3) to conduct MRV to fight against malware detection in practice.

`C1: Defeating Black-box Detector.` Conducting MRV requires the internal knowledge and understanding of malware detectors. Unfortunately, generally an adversary may have little (or even no) knowledge about the malware-detection model (such as features and algorithms). Moreover, the particular knowledge to a malware-detection model is too specific and is unlikely to succeed if the model (*e.g.,* the one in VirusTotal [6]) uses the combination of several techniques.

`C2: Preserving Malicious Behaviors.` The mutated malware should maintain the original malicious purposes and therefore simply converting malware's features to another app's features is likely to break the malicious behaviors. For example, the malicious behaviors are usually designed to be triggered under certain contexts (to avoid user attention and gain maximum profits [55]), and the controlling logic of the malware is too sophisticated (e.g., via logic bombs and specific events) to be changed.

`C3: Maintaining the Robustness of Apps.` The mutated malware should be robust enough to be installed and executed in mobile devices. Automatically mutating an app's feature values is likely to break the code structures and therefore crash the app.

Our Approach. In this paper, we propose two attacks that can conquer these challenges and produce *evasive* malware variants based on observations of two fundamental limitations of malware detection (Sec. 2.1).

We propose *Malware Evolution Attack* to mimic the feature mutations in the space of actual malware evolution in the real world. This attack is based on the observation that evolution of a malware is actually the process of mutating its corresponding feature values to reach "*blind spots*" of malware detection. For example, we can follow the feature mutations from M_1 to M_2 (*i.e.,* $c \rightarrow c'$ in f_3) to derive a new malware variant M_v from M_3 (shown in Table 1). In particular, we conduct *phylogenetic analysis* on each family of malware and derive mutation strategies based on the phylogenetic tree of each malware family. Such attack follows the idea of aforementioned strategy of look-like-unclassifiable-app (iii)[3], but as the attack is derived from the existing mutations, the likelihood of the mutations to break the malicious functionalities decreases (as confirmed by our empirical evaluation). Interestingly, we find that some of the mutation strategies effectively generate evasive malware variants when applied across families (*i.e.,* using the evolution history of a malware family to guide the mutation of malware instances from a different family).

We propose *Malware Confusion Attack* to mutate the malware features from the original feature values to the ones that are less differentiable for malware detection. This attack is based on the observation that malware detection generally performs poorly in differentiating the malware from benign apps given the same feature values. For example, in Table 1, malware M_2 shares the same feature values with the benign app B_1, and the malware detector cannot tell the difference and therefore marks them in the same category (producing false positives if the label is malware while producing false negatives if it is benign). In particular, we mutate differentiating feature values (*i.e.,* feature values that exist in *only* malware) to confusing feature values (*i.e.,* feature values that exist in *both* malware and benign apps) so that malware detection techniques have a hard time to identify the malware based on the features. This attack follows the idea of aforementioned strategy look-like-misclassified-malware (ii) but does not rely on the detection result using a particular detection model; such characteristic potentially makes all detection models vulnerable.

To address the challenge of `defeating black-box detector (C1)`, we propose a RTLD feature model that generally reflects the susceptibility of a detection technique to the mutations of malware feature values. This model is based on the insight that malware detection techniques not only leverage essential features of malicious behaviors (*e.g.*, security-sensitive method calls), but also incorporate other contextual factors (*e.g.*, when and where a method is invoked) to reflect the malicious intentions [23, 55, 58]. Additionally considering contextual factors is due to that the same values of such essential features exhibit in both malware and benign apps, so using only essential features is typically insufficient to differentiate malicious behaviors from benign ones. The proposed RTLD feature model summarizes both essen-

[3]Note that we do not come up with any attack that conforms to aforementioned strategy of look-like-benign-app (i). The main reason is that, based on our empirical results, using features (*e.g.,* program structure) that exist in only benign apps is likely to break malicious behaviors and eliminate the maliciousness.

tial and contextual features in mobile apps, and categorizes them in four aspects (*i.e.,* resource, temporal, locale, and dependency). Note that as a proof of concept, the RTLD model used in this paper is only an approximation of feature space used in malware detectors, so it is likely that the RTLD model would miss some features and thus MRV neglects to mutate those features. Our empirical results (Sec. 10) show that MRV based on our RTLD model is already able to mutate a non-trivial portion of malware samples, demonstrating the effectiveness of the model. Actually, we expect that MRV would produce better result given a more comprehensive feature model.

To address the challenge of `preserving malicious behaviors (C2)`, we propose a similarity metric to help find matching contextual features for certain malicious behaviors. The metric is defined based on the likelihood of two methods to appear in the same program basic block (*i.e.,* under the same execution context). Such metric ensures that the mutated contextual features of a malicious operation (*i.e.,* the invoking context for a security-sensitive method call) must come from the contextual features of the same or similar malicious operation (based on their usage pattens in existing apps). Using the metric helps preserve the dependencies and controlling logics of malicious behaviors, making it unlikely to be broken down.

To address the challenge of `maintaining the robustness of apps (C3)`, we develop a new technique, inspired by program transplantation [39, 47], to reuse the existing implementations of desired features instead of randomly mutating or synthesizing the code. In particular, we develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation[4]. By leveraging the existing implementations, this technique enables systematic and automatic mutations on malware samples while aiming to produce a well-functioning app.

Contributions. This paper makes the following main contributions.

- `Observation.` We identify differentiability of selected features and robustness of detection models as two fundamental limitations of malware detection, from which we demonstrate the feasibility of producing effective attacks (Sec. 2).
- `Characterization.` We propose an RTLD feature model that characterizes and differentiates contextual and essential features of malicious behaviors (Sec. 3).
- `Attacks.` We propose malware recomposition variation (MRV) to produce two types of *feature evolution attack* and *feature confusion attack* to effectively mutate existing malware for evading detection (Sec. 5).

[4]Transplanting a feature in one app/component/method (*i.e.,* donor) to a different app/component/method (*i.e.,* host).

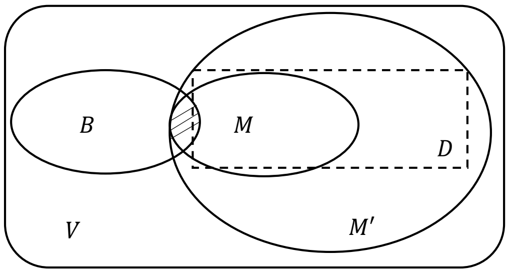

Figure 1: A feature vector space V, the feature vectors of existing benign apps B, the feature vectors of existing malware M, the feature vectors can be detected by detection model D, the feature vectors of all potential malware M' and their relationships.

- `Framework.` We develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation to automatically mutate app features (Sec. 6).

2 MRV Design

2.1 Limitation of Malware Detection

MRV leverages two fundamental limitations of malware detection: *differentiability of selected features* and *robustness of detection model*. To better illustrate the limitations, we model the vector space of features used by a malware-detection technique as V (shown in Figure 1).

The differentiability of selected features can be represented by the intersection of the feature vector space (denoted as B) for the existing benign apps and that (denoted as M) of existing malware. In an ideal case, if the selected features are perfect (*i.e.,* all differences between benign apps and malware are captured by features), no malware and benign apps should be projected to the same feature space, *i.e.,* $B \cap M = \varnothing$. Such perfect feature set, however, is difficult or even impossible to get in practice. For example, to detect a malware that loads a malicious payload at runtime, a malware detector could use the name of the payload file as a feature for the detection. Unfortunately, the name of the payload file can be easily changed to a common file name used by benign apps to evade the detection, therefore resulting in false negatives. If the detector removes such a feature in fighting malware, the detector produces false negatives by incorrectly catching benign apps that may have behaviors of dynamic code loading. In either way, the selected feature set is imperfect to differentiate such malware and benign apps.

The robustness of a detection model can be represented by the difference between the feature vectors (denoted as M') of all potential malware and the feature vectors (denoted as D) that can be detected by the detection

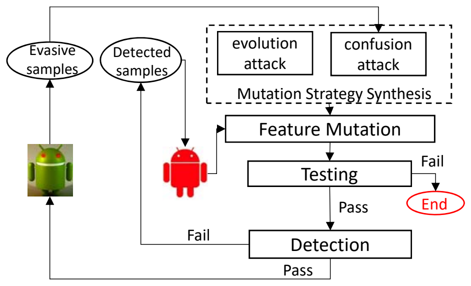

Figure 2: Illustration of mutant construction in evolution MRV. Key steps: (1) transformation-strategy synthesis; (2) program transformation/feature mutation; (3) program testing.

model[5]. Such difference can be denoted as $M' \setminus D$. A perfect detection model should detect all possible malicious feature vectors (*i.e.,* M'). In practice, detection models are limited in detecting existing malware because it is hard to predict the form of potential malware (including zero-day attacks). In this work, we argue that a robust malware-detection model should aim to detect malware variants produced through known transformation. Such transformation should employ not only syntactic and semantic obfuscation techniques, but also feature mutations from analyzing the evolutions of malware families.

2.2 Threat Model

We assume that an attacker has only black-box access to the malware detectors. Under such assumption, the attacker can feed any malware sample as the input to the detectors and know whether the sample can be detected or not, but the attacker has no internal knowledge (*e.g.,* detection model, signature, feature set, confidence score) about the detectors. The attacker is capable of manipulating the binary code of the malware, but has no access of the source code of the malware. We assume that the attacker has access to the existing malware samples (*i.e.,* samples that are correctly detected by malware detectors), and the goal of the attacker is to create malware variants with the same malicious behavior, but can evade the detection.

2.3 Overview of MRV

We propose *Malware Recomposition Variation (MRV)*, the *first* work that systematically reconstructs new types of malware using decompositions of features from existing malware to evade detection. MRV first performs *mutation-strategy synthesis* (Sec. 5) including both *feature evolution attack* and *feature confusion attack*, and then MRV leverages program transplantation to mutate the existing malware (Sec. 6) where program testing is used to find the survival app transformations[6]. MRV is built upon the RTLD feature model that captures both the essential and contextual features of malware (Sec. 3). Fig 2 illustrates the whole framework used to generate app mutants in evolution MRV (*i.e.,* Malware Evolution Attack).

In MRV, the *feature evolution attack* is proposed based on the insight that reapplying the feature mutations in malware evolution can create new malware variants that may evade detection (*i.e.,* the feature vectors fall into the area of $M' \setminus D$). As Figure 2 shows, the attack mutates RTLD feature values iteratively at each level (following the sequence of temporal feature, locale feature, and dependency feature).

In MRV, the *feature confusion attack* is proposed based on the insight that malware detection usually performs poorly in differentiating the malware and benign apps with the *same* feature vector. As discussed earlier, if we simply mutate malware features to benign feature vectors (*i.e.,* features in space B), such mutation would generally break or weaken the malicious behaviors (*i.e.,* turning the malware into benign apps). So our design decision is converting malware with unique malicious features (*i.e.,* $M \setminus (B \cap M)$) to possess the features shared with benign apps (*i.e.,* $B \cap M$). Because some malware already possess such feature vectors, we could leverage the program transplantation technique to transplant the existing implementation to the host malware. Using program transplantation greatly decreases the likelihood of breaking the original malicious behavior in the host malware. Instead of mutating the individual feature iteratively at each level, feature confusing attack mutate the whole feature vector.

Use Cases. Although our techniques are presented as attacks to malware detection, the techniques can also be used in assisting the assessment or testing of existing malware-detection techniques, to enable the iterative design of a detection system. The main idea is to launch feature evolution attack and feature confusion attack on each revision of the detection system, so that security analysts can further prune their selection of features in the next revision. *Feature evolution attack* can be used to evaluate the robustness of a detection model. The more robust the detector model is (*i.e.,* the larger D is), the more difficult for a mutated malware to evade detection (*i.e.,* the smaller $M' \setminus D$ can be). The detail of each step is elaborated in subsequent sections. *Feature confusion attack* can be used to evaluate the differentiability of se-

[5]We safely assume that for a reasonable malware-detection model, $D \subseteq M'$. A reasonable malware-detection model produces false positives on a benign app only because the feature vector of the benign app is shared by some malware.

[6]Note that in practice, when the transformed apps fail (either crash or cannot evade detection), we iterate the program transformation process by mutating other contextual feature values. Fortunately, in real-world malware, there are many feature types that allow to be mutated differently.

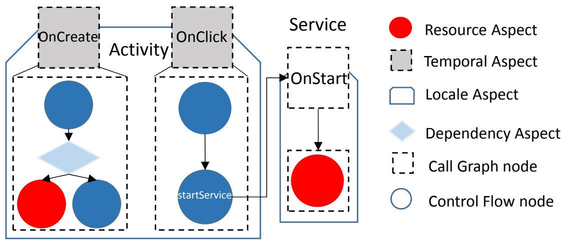

Figure 3: RTLD features in an Android program

lected features. The more differentiable a feature is, the less the opportunity is for a malware to confuse the detector (*i.e.,* smaller $B \cap M$ is desirable).

3 RTLD Feature Model

In this work, we characterize semantic features using proposed RTLD feature model that aims to reflect the essential malicious behaviors while making a balance between the computational efficiency and accuracy (shown in Figure 3). RTLD feature is designed based on a simple observation: *the invocation of malicious behaviors (and associated payload) depends on the condition of the malicious code execution, i.e., when a particular malicious behavior is trigged out in certain context.* For example, an SMS message sending behavior is malicious only when it is performed in the background stealthily without notifying the users, and also the message itself contains the users' sensitive information, such as location, identify, etc. In contract, if an SMS message is sent publicly without any sensitive information leakage, it should be labeled as a benign behavior. The novel semantic feature (Figure 3) covers four different aspects:

• **Resource aspect.** What security-sensitive resources obtained by malicious behaviors?

• **Temporal aspect.** When the malicious behaviors are triggered?

• **Locale aspect.** Where the malicious behaviors occur?

• **Dependency aspect.** How the malicious behaviors are controlled?

Resource aspect features describe the security-sensitive resources exploited by malicious behaviors while the dependency features further represents how the malicious behaviors are controlled. In practice, they are closely linked to finish one task, and therefore, we present them together according to different malicious behavior types.

• *Access to permission-protected resources.* Permission-protected resources is essential to malicious behaviors. Malware usually use Android APIs to get the privileges or sensitive information. Therefore we leverage static analysis techniques [11, 28, 49] to find locations of the API method invocations (*e.g.,*, red circle in Figure 3).

• *Execution of external binaries/commands.* Malware can circumvent permission system to access security-sensitive resources. Instead of invoking permission-protected APIs, malware bypass the system services by performing root exploit and command injection exploits. By doing this, malware escalate privileges and continue the next operations (*e.g.,* turning the phone into a botnet or installation of other malware [7]). We summarize the individual methods used in root exploits (e.g., Runtime.exec, native methods) and identify their use by static analysis.

Temporal aspect features describe the contexts when the malicious behaviors are triggered. In particular, we characterizes the events triggering malicious behaviors as follows.

• *System events.* System events are initiated by the system state changes (*e.g.,* receiving SMS and phone boots) and broadcast to all apps. Apps can register a broadcast receiver to receive specific system events.

• *UI events.* UI events are triggered by user interactions on apps' UI, such as clicking a button or scrolling a list view. Apps can register an event handler to receiver UI events.

• *Lifecycle events.* Lifecycle events are user interactions on interfaces of systems or devices that can change the lifecycle of an Android component. Android apps handle such events by implementing life cycle methods for Android components. However, not every lifecycle method corresponds to a lifecycle event. In many cases, lifecycle methods are invoked in response to system events or UI events. In these cases, we classify the activation events of permission uses as system events or UI events. The defined lifecycle events are handled by code in the underlying system, rather than code in apps (*e.g.,* events triggered by the interactions on the device interfaces such as pressing the `HOME` or `BACK` button).

Locale aspect features describe the component where the malicious behavior occurs. The location of the execution is either an Android component (*i.e.,* `Service` , `Activity` and `Broadcast Receiver`)or concurrency constructs (*e.g.,* `AsyncTask` and `Handler`). The malicious behaviors get executed when these components are activated. Due to the inter-component communication (ICC) in Android program, the entrypoint component of a malicious behavior could be different from the component where the behavior resides in.

The locale features in general reflect the visibility (*i.e.,* whether the execution of the task is in the foreground or background) and continuity (*i.e.,* whether the task is once-ok execution or a continuous execution (even after

[7]For example, `DroidKungFu1` first roots the phone and then installs the malicious payload to turn the phone into a botnet. The resources exploited for this behavior are Java class runtime and external rootkit binaries (*e.g.,* "exploid", "rageagainstthecage").

```
public class User extends Application{
public String androidid;
public String tel;}
```

(a) `User` class of DougaLeaker malware

```
public class MainActivity extends Activity{
public void onCreate(android.os.Bundle b){
super.onCreate(b);
this.requestWindowFeature(1);
User u = (User) getApplication();
u.androidid = Settings.Secure.getString(
        getContentResolver(), "android_id");
u.tel = getSystemService("phone").
        getLine1Number();
if(isRegisterd(u.androidid)){
Cursor cursor = managedQuery(ContactsContract.
        Contacts.CONTENT_URI, 0, 0, 0, 0);
while (cursor.moveToNext() != 0) {
this.id = cursor.getString(cursor.getColumnIndex("
        _id"));
this.name = cursor.getString(cursor.
        getColumnIndex("display_name"));
this.data = new StringBuilder(String.valueOf(this.
        data)).append("name:").append(this.name).
        toString();
}
cursor.close();
}else{
startService(new Intent(getBaseContext(),
        MyService.class));
}
}
this.exec_post(this.data);}} //sending contacts
        through HttpPost
```

(b) `MainActivity` of DougaLeaker malware

```
public class MyService extends Service{
public int onStartCommand(Intent intent, int flags,
         int startId){
User u = (User) getApplication();
String text = "android_id = " + u.androidid + ";
        tel =" + u.tel;
Date date = new Date();
if(date.getHours>23 || date.getHours< 5 ){
android.telephony.SmsManager.getDefault().
        sendTextMessage(this.number, null, text, null,
         null);
}
return;}}
```

(c) `MyService` of DougaLeaker malware

Figure 4: Motivating Example: DougaLeaker malware

exiting the app)). For example, if a permission is used in a Service component (that has not been terminated by `stopService`), then the permission use is running in the background, and also it is a continual task (even after exiting the app).

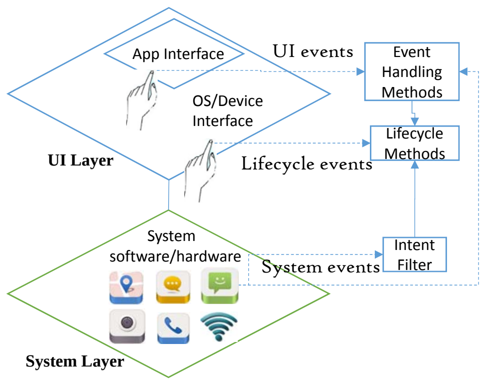

Figure 5: An illustration of events in Android Apps

Dependency aspect features describe the control dependencies of the invocation of the malicious behaviors. A control dependency between two statements exists if the truth value of the first statement controls whether the second statement get invoked. For example, in Listing 1, the invocation of `sendTextMessage` is controlled by the truth value of `planeMode`. Malware frequently leverage external events or attributes to control the malicious behaviors. For example, `DroidDream` leverages current system time to control the execution of its malicious payload. It suppresses its malicious payload during the day but allows the payload executions at late night when users are likely to be sleeping.

Usually, in malware detection, such dependencies are represented by external events or attributes that can affect control flows to the security-sensitive behaviors. For example, in Listing 1, the dependence feature of `sendTextMessage` is the click action that toggles the value of `planeMode`. The control dependency features are generated from program dependence graphs (PDGs) obtained in static analysis of apps.

Listing 1: Example of control dependency

```
//Toggle airplane mode
boolean planeMode = false;
void onClick(){
  planeMode = true;
}

void sendTextMessage(String text){
  if(!planeMode){
    android.telephony.SmsManager.getDefault().
          sendTextMessage(text);
  }
}
```

4 Feature Extraction

Resource features. To extract RTLD features, MRV will locate the resource feature (i.e., security-sensitive behavior) in the program and the call graph entry points that will trigger the behavior. The purpose of locating entry points is to provide preliminary context feature information, i.e., preliminary information about when the resource feature will be used. The declared (permission-protected) resource in an Android app will be used by three kinds of operations in the app to pass permission check. First, using some of the Android API will invoke the permission checking function *checkpermission*. Second, sending or receiving Intents with certain action fields will require permissions. Last, accessing the content providers containing private information require the permissions specified by the content providers. For each of the operation, our approach identify the location of the resource feature in the program and the entry points of the app that will trigger the resource feature. Then MRV use the mapping provided by PScout to map from program artifact (i.e., API method signature, Intent action name, URI of content provider) to corresponding permissions.

To identify the APIs requiring permission checks in the given app, MRV first construct the call graph for the whole app. MRV leveraged the entry points defined in flowdroid [1] as the entry points of our call graph. These entry points include every Android component lifecycle methods and UI event handlers. After MRV got the call graph of the whole app, MRV traverse the call graph to check whether any API call in call graph matches the API listed in PScout mapping. In this way, MRV compute the list of API calls requiring permissions.

To locate these API calls in the call graph of whole application and better assist our later context extracting process, for each API call requiring the permission check, MRV extract the sub-call graphs that only includes the path from entrypoints to this API. To extract such sub-call graph, MRV compute the transitive closures in call graph from every entrypoint to the call-graph node representing the API call. MRV can infer from this sub-call graph all the entry points that will trigger the API call.

To identify the access to permission protected content provider, MRV first locates the calls to several methods of *ContentResolver* class, i.e., *query*, *insert*, *update*,*delete*. MRV uses the intra-procedural backward flow analysis to extract the statements that construct the URI parameters of *ContentResolver* method calls. MRV then analyzes whether these URIs are constructed from String constant or URI filed of Android SDK class. MRV checks the String constant or URI filed with PScout mapping to see if they require permissions. If permission is required, MRV constructs the sub-call graph from entrypoints to the call-graph node representing the *ContentResolver* method call.

To identify the resource feature in sending intent, MRV locate the calls to *Context.sendBroadcast()* in call graph of whole app. MRV do intra-procedural backward flow analysis to extract the statements that set *action* field of the Intent parameter of *sendBroadcast*. MRV then scan through Pscout mapping to see whether the action field require any permission. If the action require permission, MRV construct the sub-call graph from entrypoints to the call-graph node representing the *sendBroadcast*.

The resource feature for receiving intent could located both in program code and manifest file. Apps could register the intent filter by both explicitly declare intent-filter in the manifest file, or by calling *Context.registerReceiver()* method in the code. For *registerReceiver* method calls in the program code, MRV do the same analysis as *sendBroadcast*. The only difference is that MRV extract the statements of IntentFilter parameter instead of Intent. For intent filter registered in manifest file, MRV extract the manifest file from the app's APK file. Then MRV extract the action field of components' intent-filters and check if the action field require any permission. If the action require permission, MRV link the components to permissions.

Algorithm 1: resourceFeaturePoint

```
Input  : $\mathscr{A}$: App binary code
         $\mathscr{P}_a$: API methods-permissions mapping
         $\mathscr{P}_c$: Content provider URI-permission mapping
         $\mathscr{P}_i$: Intent action-permission mapping
Output: $\mathscr{B}$: A set of resource features and their entry points
1  begin
2      $WholeCG \leftarrow getWholeCallGraph(A)$
3      $PerList \leftarrow getPermissionList(A)$
       // Get list of method calls that may require
       permissions
4      $MethodCalls \leftarrow getPermissionAPIs(\mathscr{P}_a, WholeCG)$
5      foreach $\mathscr{M} \in MethodCalls$ do
6          $SubCG \leftarrow getSubCallGraph(WholeCG, \mathscr{M})$
7          $Entrypoints \leftarrow getEntrypointsFromCG(SubCG)$
8          if $isContentMethod(\mathscr{M})$ then
9              $URI \leftarrow getURI(SubCG, \mathscr{M})$
10             if $URI \in \mathscr{P}_c$ then
11                 $Perm \leftarrow getPermission(URI, \mathscr{P}_c, \mathscr{M})$
12                 $\mathscr{B}.add(Perm, URI, Entrypoints)$
13             end
14         end
15         if $isIntentMethod(\mathscr{M})$ then
16             $Action \leftarrow getIntentAction(SubCG, \mathscr{M})$
17             if $Action \in \mathscr{P}_i$ then
18                 $Perm \leftarrow getPermission(Action, \mathscr{P}_i, \mathscr{M})$
19                 $\mathscr{B}.add(Perm, Action, Entrypoints)$
20             end
21         end
22         if $isAPIMethod(\mathscr{M})$ then
23             $Perm \leftarrow getPermission(\mathscr{M}, \mathscr{P}_a, PerList)$
24             $\mathscr{B}.add(Perm, \mathscr{M}, Entrypoints)$
25         end
26     end
27 end
```

Algorithm 1 presents the analysis to infer resource features and their entry points in the call graphs. The analysis varies based on the three types of operations. First, if $\mathscr{M}$ is a method that accesses content providers, the analysis verifies whether URI parameter of $\mathscr{M}$ is listed in the URI-permission mapping ($\mathscr{P}_c$), and saves the URI, permission, and the entry points for sub-call graphs if the URI is in the mapping.(Line 7-9) Second, if $\mathscr{M}$ is a method that sends or receives intents, the analysis verifies whether value of action filed of the intent parameter in $\mathscr{M}$ is listed in the intent action-permission mapping ($\mathscr{P}_i$), and saves the intent, permission, and the entry points for sub-call graphs if the URI is in the mapping. (Line 10-12) Finally, if $\mathscr{M}$ is a method that invokes a permission protected API method, the analysis extracts and saves the method name, permission, and the entry points for sub-call graphs (Line 13-15).

Although our static analysis can locate all the resource features in the code, the resource features can also be in manifest files. Apps can register intent filters in the manifest files to receive intents requiring permissions. In this case, MRV parses the manifest file to retrieve the action fields of intent filters. If the action fields match any record in the action-permission mapping, the permissions and action fields are used to represent resource features.

Locale features Algorithm 2 presents the analysis to infer location feature from the entry points provided by the previous step. Our analysis divides the entry points into three categories. For each category, MRV takes different steps to identify execution settings.

The first category of entry points is asynchronous methods in Android (e.g., *Handler.handleMessage* and *AsyncTask.doInBackground*). Asynchronous methods run in the background for one-time task; therefore, the the continuity is set as "discontinued" (Line 8). To assist the later task of extracting temporal feature, the analysis replace asynchronous methods with the entry points (lifecycle methods) triggering asynchronous methods (Lines 9).

The second category of entry points is life cycle methods of service components. The analysis checks the existence of methods that could terminate the service in "onStop" or "onPause" methods in the call graph and the return value of its "onStartCommand" to determine if the service could be terminated and not automatically restart(Line 13). If the service meet both requirements, the permission use may be a once time task (Line 24). If not, the service runs continuously in the background (Line 25).

The last category of entry points is life cycle methods of other components. These components do not execute continued tasks (Lines 22).

Temporal features Then, MRV extracts temporal features. In this step, MRV will recover the context from the external events that trigger the resource feature. To know the external event triggering the resource feature, MRV need to locate the entry points that handling these events. MRV will first locate these entry points by identifying component that make inter-component communication. Then MRV extract the attribute of the event handling entry points to recover the context information.

Algorithm 2: identifyLocFeatures

```
Input  : $\mathscr{B}$: A set of resource features and their entry points
         $\mathscr{A}$: App binary code
Output: $\mathscr{E}$: A set of resource features, location features with entry points
         of their temporal features
1  begin
2      $WholeCG \leftarrow getWholeCallGraph(A)$
3      foreach $b \in \mathscr{B}$ do
4          $entrypoints \leftarrow getEntrypoints(b)$
5          foreach $entrypoint \in entrypoints$ do
6              $c \leftarrow newContext()$
7              $c.setInvocation(\mathscr{B})$
8              if $isAsyncCall(entrypoint)$ then
9                  $c.setVisibility("background")$
10                 $c.setContinuity("once")$
11                 $\mathscr{M} \leftarrow findAsyncCall(entrypoint, \mathscr{A})$
12                 foreach $m \in \mathscr{M}$ do
13                     $G \leftarrow getSubCallGraph(WholeCG, m)$
14                     $E \leftarrow getEntrypointsFromCG(G)$
15                     foreach $e \in E$ do
16                         $c_1 \leftarrow c\ c_1.setEntypoint(e)$
17                         $\mathscr{E}.add(c_1)$
18                     end
19                 end
20             end
21             if $isService(entrypoint)$ then
22                 $c.setVisibility("background")$
23                 if $isTeminate(entrypoint, WholeCG)$ then
24                     $c.setContinuity("once")$
25                 end
26                 else
27                     $c.setContinuity("continuous")$
28                 end
29                 $\mathscr{E}.add(c)$
30             end
31             if $isReceiver(entrypoint)$ then
32                 $c.setVisibility("background")$
33                 $c.setContinuity("once")$
34                 $\mathscr{E}.add(c)$
35             end
36             else
37                 $c.setVisibility("foreground")$
38                 $c.setContinuity("once")$
39                 $\mathscr{E}.add(c)$
40             end
41         end
42     end
43 end
```

The entry points of the call graph may not necessarily be the entry point of the program because of the inter-component communication inside an app. For example if the entry point of the call graph is the *onCreate* method of an Android component, the *onCreate* could be triggered by the method call (e.g., *startActivity*, *startService*) inside the app. In this case, MRV trace to the component that triggered *onCreate* to find the external triggering event.

To identify the inter-component communication, MRV first manually summarize the inter-component communication method (e.g., *startActivity*, *startService*, *startActivityForResult* etc.) MRV scan through the call graph of the whole app to locate the call site of these methods. Then, MRV do intra-procedural backward flow analysis to collect the statements that set the component attribute of Intent. By parsing the component attribute MRV get the target component name of the inter-component communication. MRV construct altered call graphs for inter-component communication methods. MRV altered the name of call node of inter-component communication method by adding the target component name. Thus, MRV can differentiate the call site of methods that invoking different component. These altered call graphs present the entry points for inter-component communication methods. And MRV use these graph to find the event handling entry points by algorithm 3.

It's worth noted that some components can be invoked by both inter-application intent message and intra-application intent message. For those components, we consider both the *onCreate* method of itself and the entry points of its invoking component as the entry points to handle external events.

As mentioned earlier, the temporal features can be classified into four categories based on the handling mechanism for the temporal features. MRV identifies all four categories of temporal features by the attributes of their entry points. (1) For system events handled by intent filters, their entry points are life cycle methods. The components of the life cycle methods should have intent filters specified. (2) For both system events captured by event-handling methods and (3) UI events, their entry points should be event-handling methods. (4) For life cycle events, their entry points are life cycle methods, and these life cycle methods have not been invoked by other events. MRV does not analyze the resource features in manifest files in this step, because the temporal features of these resource features are the resource features themselves.

Algorithm 3 presents the analysis to extract temporal features for the given permission uses. The analysis returns a list of temporal features ($Contexts$) for each permission use, because Android apps often provide different ways for users to invoke the same functionality. The analysis takes entry points of an locale feature as input. The entry point could be of only two categories: lifecycle methods and event-handling methods.

For the first category of entry points, lifecycle methods, the analysis first decides whether the temporal features could be the system events captured by intent filters (Line 4). If the component of the lifecycle method has intent filters, for each of the intent filters, the attributes of the intent filter are used to represent a context. The generated contexts for system events are then saved to the $Contexts$ list (Line 7).

The analysis then decides whether the lifecycle method can be invoked from the code (startService, sendBroadcast, etc.) instead of intent filters (Lines 10-11). If there are method calls invoking the life cycle method, the analysis constructs the sub-call graph (*subCG*) for each method m of the method calls (Line 13). Then the entry point for the *subCG* is passed to Algorithm 3 to get the contexts recursively (Line 16). By calling Algorithm 3, these contexts use the temporal features of invoking method m as their temporal features. The returned contexts are saved to the $Contexts$ list (Line 17).

If the life cycle method cannot be invoked from program code, then the permission use is triggered by lifecycle events. We use the attributes of the lifecycle methods to represent the temporal feature of the context, and save the context for lifecycle events to the $Contexts$ list (Line 22).

For the second category of entry points, event-handling methods, the analysis uses the attributes of the UI event-handling methods or system event-handling methods to represent the temporal feature of the context, and save the context to $Contexts$ list (Lines 26).

Algorithm 3: setTemporalFeatures

```
Input  : e: A set of resource features, location features with entry points
         of their temporal features
         entrypoint: entrypoint of features,
         𝒜: App binary code
Output: Contexts: List of RTL features for e
1  begin
2      Contexts ← ∅
3      if isLifeCycleMethods(entrypoint) then
4          if hasIntentFilters(entrypoint, A) then
               // System events (by intent filters)
5              Filters ← getFilters(entrypoint, A)
6              foreach filter ∈ Filters do
7                  c ← e
8                  c.setFilterAttributes(filter)
9                  Contexts.add(c)
10             end
11         end
12         WholeCG ← getWholeCallGraph(A)
13         ℳ ← findInvokingMethods(WholeCG, entrypoint)
14         if ℳ ≠ ∅ then
15             foreach m ∈ ℳ do
16                 SubCG ← getSubCallGraph(WholeCG, m)
17                 E ← getEntrypointsFromCG(SubCG)
18                 foreach entry ∈ E do
19                     cts ← setTriggeringEvent(entry, e, A)
20                     Contexts.add(cts)
21                 end
22             end
23         end
24         else if ¬ hasIntentFilters(entrypoint, Manifest) then
               // LifeCycle events
25             c ← e
26             c.setLifeCycleAttributes(entrypoints)
27             Contexts.add(c)
28         end
29     end
30     if isUIEventHandler(entrypoint) then
           // UI Events
31         c ← e
32         c.setUIEventsAttributes(entrypoint)
33         Contexts.add(c)
34     end
35     if isSystemEventHandler(entrypoint) then
           // System events (by event handling methods)
36         c ← e
37         c.setSystemEventsAttributes(entrypoint)
38         Contexts.add(c)
39     end
40     return c
41 end
```

Dependency features After computing RTL features, MRV constructs control-flow graph (CFG) to extract dependency features. For each resource features, MRV identifies the entry points that can lead to the invocation of the resource method. Then MRV obtains the inter-procedural control-flow graph (ICFG) of the app by connecting the CFG of each node on the ECG. Based on the ICFG, MRV constructs the subgraphs from each entry point to the security-sensitive method call. For each subgraph, MRV traverses the subgraph to identify the conditional statements on which the security-sensitive method is control-dependent. Finally, MRV saves the set of extracted conditional statements as dependency features with the resource features and the corresponding location/temporal features.

5 Mutation strategy synthesis

We present our techniques regarding synthesize strategies to mutate program features that achieve two goals.

• Goal 1: the mutation strategy should enable the mutated malware to evade the detection of existing techniques (*i.e.*, reach "blind spot" of detectors).

• Goal 2: the mutation strategy should preserve the malicious behaviors of the original malware (*i.e.*, achieve the same malicious functionality without crashing).

To achieve the aforementioned goals, we develop: (i)*black-box scenarios* where adversaries only know the binary detection result (*i.e.,* malicious or not) without any additional information; (ii) *informative scenarios* where adversaries know the type of the algorithm used in detection. The mutation strategies, if automated, can be systematically employed on large scale samples, enabling the exploitation to identify more blind spots of existing detection and generation of more targeted variants that can achieve malicious purposes while evading detection.

5.1 Evolution Attack vs. Confusion Attack

In *black-box scenarios*, adversaries have no knowledge about malware detection techniques (*e.g.,* features, models, algorithms). So instead of developing targeted malware to evade specific detection techniques, we come up with a more general defeating mechanism called **evolution attack**: *mimicking and automating the evolution of malware*. Such defeating mechanism is based on the insight that the evolution process of malware reflects the strategies employed by malware authors to achieve a malicious purpose while evading detection. Although existing anti-virus techniques have already been updated to detect the "blind spot" exploited by evolved malware samples, those malware samples are merely a few instances being manually mutated by malware authors. The core idea of malware evolution attack is to create new malware variants based on the analysis of malware evolution (*i.e.,* copy and edit of the patterns) so as to evade detection. The new generated variants are "blind spot" of current detectors.

In *informative scenarios*, adversaries know the type of algorithms used in the detection, and therefore we develop the targeted attack called **confusion attack**. The main idea of malware confusion attack is to mimic the malware that can generally evade detection, *i.e.,* confusing the malware detectors by modifying the feature values that can be shared by malware and benign files. Such defeating mechanism is based on the observation that malware detector (based on a classifier) could be easily misled by the feature omissions and substitutions.

5.2 Algorithm Details

In both attacks, to find the features that can cause confusion and evolution, we first extract RTLD features corresponding to each malicious family and benign category (*i.e.,* projecting all apps to RTLD feature spaces).

(i) To generate **evolution attack**, we identify a feature set called *evolution feature set*. In the set, each feature is evolved either at intra-family level or inter-family level. For each feature vector in the *evolution feature set*, we count the number of evolutions as *evolution weight*, where *intra-family evolution weight* is proportional to the number of evolutions at intra-family level and *inter-family evolution weight* is proportional to that at inter-family level. The rationality is that if the feature type has already been evolved frequently in observation, it is more likely to be evolved according to the nature of the law (in biological evolution process [12]). Figure. 6 demonstrates how 32 samples are evolved in `AdDisplay` family [8].

(ii) To generate **confusion attack**, we identify a set of feature vectors that can be projected from both benign apps and malware as *confusion feature set*. For each feature in the confusion feature set, we count the number of benign apps that can be projected to the feature vector as the *confusion weight* of the feature vector. The rationale is that if more benign apps are projected to the feature, it is harder for the malware detector to label the apps with this feature as malicious.

After the above step, in both **evolution attack** and **confusion attack**, for each malware that we aim to mutate, we first check whether the resource feature appears in any *critical feature set* that denotes *evolution feature set* in the context of *evolution attack* and *confusion feature set* in the context of *confusion attack* respectively.

(iii) If a resource feature R appears in a vector V in the critical feature set, we then mutate the original feature vector of R to be the same as the vector V by mutating

[8]The number labeled in the bottom of each phylogenetic tree denotes the distance between two nodes. The node could be a leaf node that denotes a malware, and also could be an internal node that denotes a cluster grouped from its children node. Hungarian-type Algorithm [37] is used to compute the malware distance regarding RTLD features.

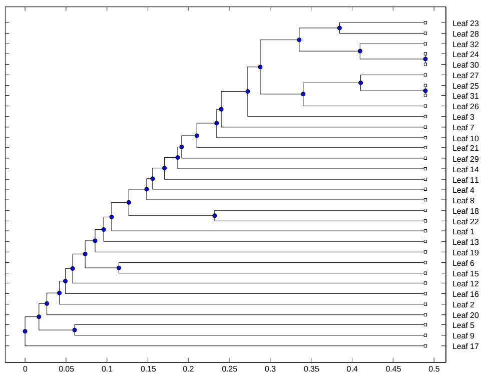

Figure 6: Phylogentic tree generated for `addisplay` family. Each leaf in the graph denotes a malware in `addisplay` family, where leaf nodes (1–9) belong to `addisplay.adswo`, (10–23) belong to `addisplay.airpush`, (24–25) belong to `addisplay.dowgin`, (26) belongs to `addisplay.kuguo` , (27–28) belong to `addisplay.waps`, (29–32) belong to `addisplay.wooboo`.

the contextual features. A resource feature could appear in many vectors in the critical feature set. Then we mutate top K matching vectors ranked by the corresponding evolution or confusion weight.

(iv) Otherwise, we leverage *a similarity metric* to find another resource feature (in the critical feature set) R' that is most likely to be executed in the same context as R. Similarly we select top K vectors (ranked by the corresponding evolution or confusion weight) matching R' as the target vectors for mutation.

Finally, if any mutated malware passes the validation test (Section 7) and evades detection, then evolution/-confusion attack successfully produces a malware variant given the fact that each malware is generally corresponding to multiple mutated malware. Empirically we set $K = 10$ in experiments.

The similarity metric is given by the likelihood that two security-sensitive methods (*i.e.,* resource features) resides in the same program basic block. For each security-sensitive method m appearing in the confusion vector set, we count the number of its occurrences O_m in all apps. Similarly, for another security-sensitive method n that appears at least once in the same basic block as m, we count the co-occurrences of m and n in the same basic block as O_{mn}. Then the likelihood that method n is invoked under the same context as method m is given by: $S_{mn} = \frac{O_{mn}}{O_m}$. Such equation suggests the likelihood that m's context is compatible with that of n. Go back to case (iv), for any security-sensitive method n that does not appear in the *critical feature set*, we can select method m_i: (a) appearing in critical feature set, (b) having the highest similarity score $S_{m_i n}$; as the one probably to be executed in the same context as n.

6 Program Mutation

Figure 3 presents an example where the dotted squares represent call graph nodes and solid circles represent an abstraction of control flow graph nodes and each aspect in RTLD can be represented as a constituent part of the program. The process of mutation is essentially a transplantation process that keeps the content in resource aspect (red solid circle), while changing the contexts of resource aspect (*i.e.,* other contextual aspects). To change the contexts of resource aspect, we develop a program transplantation framework that satisfies two needs: (a) transplanting the malicious behavior (*i.e.,* resource aspect) to different contexts in the existing program; (b) transplanting the contextual features from other programs into the existing contexts.

6.1 Transplantation framework

Transplantation is the process that transplants the implementation of a feature (*i.e.,* organ) from one app (*i.e.,* donor app) to another app (*i.e.,* host app) [10].We broaden the concept of transplantation to components and methods in the same app. Transplantation takes four steps: identification of the organ (*i.e.,* code area that needs to be transplanted), extraction of the organ, identification of the insertion point in the host and adaption the organ to host's environment.

In our transplantation framework, we take different strategies based on the type of features that need to be mutated. On the one hand, to change the temporal features or locale features of the program, due to the fact that the dependency of the malicious behavior to the program is usually simple to resolve, we will identify or construct a suitable context (that satisfies the targeted value of temporal features or locale features) in the existing program, then transplant the malicious behavior (*i.e.,* resource feature) to the identified or constructed location. On the other hand, to alter the dependency features of the program, because dependency features such as external attributes usually require sophisticated ways (*i.e.,* method sequence) to achieve the specific desired control, we transplant existing implementation of such control (*i.e.,* organ) from a donor app to the host app.

Such two-strategy design aims to simplify the existing software transplantation problem. In the first strategy, the transplantation is actually intra-app. We can simply save and pass the unresolved dependency and contextual information (*e.g.,* values of parameters) in the program. In the second strategy, although the transplantation is inter-app, we just need to transplant a program slice that contains one single dependency. Such transplantation is lightweight comparing to the whole implementation of a functional feature. Intra-app transplantation is feasible to temporal and locale features because synthesizing a new entrypoint or a new component within an existing Android program results in little or no impact to other areas of the program. Mutation of dependency features requires inter-app transplantation because synthesizing new dependencies within the program is challenging. The tight coupling of dependencies brings huge impact to other program behaviors and likely crashes the mutated program.

Note that although temporal features and local features all require the transplantation of the malicious behaviors, the donor (*i.e.,* area of code) that requires transplantation is different. The related code of a malicious behavior can be separated as triggering part and execution part. These two parts may not be in the same component. For example, in Figure 3, related code of malicious behavior represented by red solid circle on the right can be separated as the triggering part in `OnClick` method of activity component and execution part in `OnStart` method of service component. To mutate temporal features, the donor to be transplanted is the triggering part. To mutate locale features, the donor to be transplanted is the execution part.

We categorize the transplantation based on the locality into three levels: inter-method, inter-component and inter-program transplantation, which are illustrated below.

6.2 Inter-method transplantation

Inter-method translation refers to the migration of malicious behaviors (*i.e.,* resource features) from a method to another method in the same component. Such transplantation is usually used to change the temporal features. The organ that needs to be transplanted for temporal feature is the entry of the malicious behavior and its dependencies (*e.g.,* `startService` in Figure 3). In order to locate the entry of the malicious behavior, we construct call-graphs from the `entrypoint` of the program (corresponding to the feature to be mutated) to the malicious method call.

We then mark the node directly connected to the entrypoint on the call graph as the entry of the malicious behavior. For example, Listing 2 shows a code snippet of DougaLeaker malware [9], we locate the malicious method call `getLine1Number` and its entrypoint is `onCreate` method.

Listing 2: Code snippet in malware DougaLeaker

```
1  public void onCreate(android.os.Bundle b)
2  {
3    super.onCreate(b);
4    this.requestWindowFeature(1);
5    String androidid = Settings.Secure.getString(
         getContentResolver(), "android_id");
6    String tel = getSystemService("phone").
         getLine1Number();
7    Cursor cursor = managedQuery(ContactsContract.
         Contacts.CONTENT_URI, 0, 0, 0, 0);
8    while (cursor.moveToNext() != 0) {
9      this.id = cursor.getString(cursor.getColumnIndex("
         _id"));
10     this.name = cursor.getString(cursor.
         getColumnIndex("display_name"));
11     this.data = new StringBuilder(String.valueOf(this.
         data)).append("name:").append(this.name).
         toString();
12   }
13   cursor.close();
14   this.exec_post(androidid, tel);
15   return;
16 }
```

Then, we extract all dependency related to the entry. For example, in Listing 2, method `exec_post` (Line 14) depends on `getLine1Number`. When transplanting invocation of `getLine1Number`, the method `exec_post` needs to be transplanted together. To ensure the entry method to be invoked under the same context (*e.g.,* parameter values), we perform a backward slicing from the entry method until we reach the entrypoint of the program. For example, in Listing 2, `exec_post` uses `androidid`, which means MRV will include Line 5 into the program slice during backward slicing. Next, we create an entrypoint method that can provide temporal features that we need. The entrypoint creation is done by either registering an event handler for system or UI events or creating a lifecycle method in the component. We also edit the manifest file to register receiver components for some of system events.

To identify the insertion point in the host method, we consult the Android lifecycle model to determine the invocation sequence between donor method and host method. If the donor method is invoked before host method based on the lifecycle model, the insertion point is the beginning of the host method. Otherwise, we set the insertion point as the end of the host method.

To migrate the extracted organ to the host method, we develop a regenerator that reads the code of the organ and writes the semantically equivalent code to the host method. The regenerator handles the potential conflicts with the existing code in the host method (*e.g.,* renaming). Sometimes the extracted organ still has dependencies with the parameters of the donor method. In these cases, the regenerator will create a set of global variables to store parameter values and make the organ dependent on the global variables.

Finally, we need to remove the organ from donor methods. If some of statements are dependent on organ, the removal can cause the donor method to crash. To avoid the side-effects of the removal, we initialize a

[9]MD5 of the malware is e65abc856458f0c8b34308b9358884512f28bea31fc6e326f6c1078058c05fb9

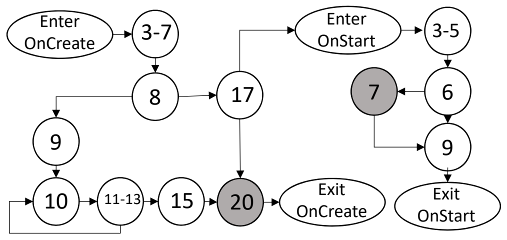

Figure 7: Inter-Procedural Control Flow Graph of DougaLeaker

set of global variables with the local variables in the organ. We then replace the original dependencies on organ by making the statements dependent on the new set of global variables. We noted that in some instances, the host method is invoked after donor method, so the set of global variables may not be initialized when donor method is invoked. So when replacing the dependencies, we add conditional statements to check for null to avoid `NullPointerException` in donor method.

6.3 Inter-component transplantation

The inter-component transplantation migrates malicious behaviors from one component to another component in the same app. Inter-component transplantation can be used to mutate the values of temporal features and locale features.

Inter-component transplantation follows the same process as inter-method transplantation except for two differences. First, in addition to temporal features, inter-component transplantation is also used to mutate locale features. As previously mentioned, to mutate local features, the organ to be transplanted is the execution part of the code. To extract such organ, we find the call graph node directly linked by the entrypoint of the execution part. Note that the entrypoint of the execution part can be different from the entrypoint of the malicious behavior. For example, in Figure 3, the entrypoint of the execution part is `onStart`, while the entrypoint of the malicious behavior is `onClick`. After we locate the call graph node, the rest of the extraction process is the same.

The other difference of inter-component transplantation is when mutating the locale feature but to maintain the temporal feature, the regenerator needs to create inter-component communications to invoke the host method. To avoid crash caused by unmatching intent messages, the regnerator will also add conditional statements to avoid executing the existing code in the host method when such inter-component communications occur.

6.4 Inter-program transplantation

The inter-program transplantation is used to migrate the dependency feature of a malicious behavior in the donor app to the host app with identical malicious behavior. The extraction of the dependency feature is different from migration of the triggering/execution part of malicious code. The organ consists of two parts. The *first* part is the implementation of the controlling behavior. We first construct the inter-component control flow graph of the app. Then we compute the subgraph containing all path from the controlling statement (*i.e.,* the statement whose value determines the invocation of the malicious behavior) to the controlled statement (*i.e.,* malicious behavior). Such subgraph essentially represents the controlling behavior. The *second* part of the organ is the dependencies of controlling statement. To extract these necessary dependencies, we slice backward from the controlling statement until we reach the entrypoint of the program. We then migrate both parts of the organ into the host app.

7 Testing on mutated apps

We perform testing on mutated apps for two purpose: (a) whether the malicious behaviors have been preserved; (b) whether the robusteness of the app has been mutated.

Checking the preserving of malicious behaviors. Checking the consistency of malicious behaviors can be quite challenging due to two reasons. First, the malicious behaviors are often triggered under specific contexts, e.g., after certain chain of user-triggered events, at certain time period, or under certain system events (*e.g.,* boot complete, receiving SMS). Second, the preserving of malicious behaviors can be hard to verify because no test oracle is available to determine the consistency between the behaviors before and after mutation.

To address those challenges, we utilize two techniques to assist the testing. First, to simulate the environment where the malicious behaviors being invoked, we create environmental dependencies by changing emulator settings or using mock objects/events. By simulating the environment, we can directly invoke the malicious behaviors to speed up the validation process. Second, to further validate the consistency of malicious behaviors when the triggering conditions are satisfied, we apply instrumentation technique to insert logging functions at the locations of malicious method calls. The logging functions print out detailed information about the variables, functions, and events invoked after the triggering events. We therefore can get the log file before and after the mutation under the same context (*e.g.,* same UI or system events and same inputs). Then, we automatically compare the two log files to check the consistency of malicious behaviors.

Checking the robustness of mutated apps. We leverage random testing to check the robustness of the mutated apps. In particular, we use Monkey [40], a random user-event-stream generator for Android, to generate UI test sequence for mutated apps. Each mutated app was test against 5,000 events randomly generated by Monkey to ensure that the app does not crash [10] .

8 Tool Implementation

In this section, we briefly illustrate implementation details of MRV. The implementation is two folds: static analysis of app binary code and constructing contexts based on static analysis result.

The static analysis of MRV is based on Soot [49]. Soot can construct call graph using binary code of apps, as its submodule Dexpler [11] converts Dalvik bytecode to Soot Jimple representation. We further leverage FlowDroid [28], a static taint analysis tool based on Soot to build our call graphs, as Flowdroid provide modeling of Android lifecycles and callback methods.

In our approach, several processes (*getURI*, *getIntentAction*, *findAsyncCall*, *findInvokingMethods*) require parameter values of the method invocations. MRV uses intra-procedural backward flow analysis to collect the statements that could affect the parameter value. Then, the MRV identify the statement that set the parameter value by method signatures. As most of the values of interests are constant values, MRV extracts the value directly from the statement by value index.

In the process of constructing contexts, MRV use mappings provided by PScout [8] as inputs. An interesting observations is that no malware is sending permission-protected intents. This is because the actions listed in PScount mapping for sending intents are often require SuperUser privileges. These intents are only allowed to be sent by apps share the same signatures with the Android systems.

The limitations of our implementation lies in two aspects. First, MRV is confined to the limitation of static analysis. Static analysis extracts what can happen rather than what does happen. Therefore, MRV may produce false positives. We also incorporate a few heuristics into our static analysis to infer the invoking relationship not shown in the call graphs. This will lead to imprecision as some of the invocation behaviors (e.g., *stopService*, *sendMessage*) are not deterministic. MRV use limited intra-procedual constant propagation. This will lead to imprecision when inferring parameter values for method invocation, especially for inferring URI strings. The Dalvik to Jimple conversion in Dexpler could also lead to imprecision, as we already confirm one case in our experiment. Second, MRV use the lists of method signatures for to identify corresponding resource, temporal, dependency methods. As some of the lists are manually collected by browsing the official Android API documents, MRV may not be able to identify some of the hidden API[11] in Android platform. For example, some apps call methods in hidden class *SqliteWrapper* to access content providers. We would miss these resource features if the list of methods accessing content providers only contain official APIs. We compensate this limitation by making MRV throw warning each time a potential resource feature is not identified. Then, we manually identify the method and add it to the lists. In future work, we plan to systematically compiling these lists by analyzing Android system code.

[10]Due to the limitation of the coverage of random testing, the mutated app passing the testing step can still be invalid. As the future work, we plan to incorporate more intelligence-guided testing techniques [29, 54] in MRV testing.

9 Experiment

Implementation. We leverage Soot [49] as our underlying static analysis framework. Soot's submodule Dexpler [11] is used to convert Dalvik bytecode into the Jimple intermediate representation from which we perform feature extraction and feature mutation. We also leverage FlowDroid [28], a static taint analysis tool based on Soot, to provide a precise modeling of the Android component life cycles and callback methods. Note that the current version of FlowDroid neither can handle native code nor can resolve reflective invocations in Java code when constructing the call graph; therefore, RTLD features in these cases cannot be extracted.

Dataset. Our subject set consists of a malware dataset and a benign app dataset. Our malware dataset starts with 3,000 malware randomly selected from Genome [59] Contagio [4], VirusShare [5], and Drebin [7]. We use VirusTotal [6] to perform sanity checking on the malware dataset (descriptions about signature-based detectors are provided later in this paper). We exclude the apps identified as benign by VirusTotal from the malware dataset. We also exclude any duplicate apps by comparing SHA1 hashes. For benign apps, we download the most popular 120 apps from each category of apps in the Google Play store as of February 2015 and collect 3240 apps in total. The process of extracting RTLD features is implemented using third-party static analysis frameworks, including Soot [49] and FlowDroid [47]. To isolate and remove the effects of potential limitations of these frameworks, we run feature-extraction analysis on the complete subject set and remove any apps that cause a third-party tool to fail. The filtering gives us a final analyzable dataset of 1917 malware and 1935 benign apps. Given the large number of apps being excluded, we reassess the distribution of our final dataset. Our final malware dataset consists of 529 malware from Genome, 25 malware from Contagio, 287 malware from VirusShare, and 1076 malware from Drebin dataset. Our final benign app dataset

[11]https://devmaze.wordpress.com/2011/01/18/using-com-android-internal-part-1-introduction/

retains 63 to 96 apps from the original 120 apps in each Google Play category. All runs of our process of extracting RTLD features, the transplantation framework, and learning-based detection tools [7, 55] are performed on a server with Intel(R) Xeon(R) CPU 2.80GH with 38 processors and 80 GB of memory with a timeout of 80 minutes in total for processing 1917 malware and 1935 benign apps.

Baseline Approaches. We implement two baseline approaches for comparison with MRV: *random MRV* and `OCTOPUS`. We first develop a random transformation strategy (random MRV) to compare against confusion and evolution attacks. Instead of following the evolution rules and similarity metrics to change the RTLD features, we randomly mutate RTLD features (*i.e.,* mutate the original feature value to the same-level feature value randomly selected from the available dataset) and transform the malware samples based on such mutation. Note that for random MRV and evolution MRV, we follow the sequence of temporal feature, locale feature, and dependency feature to apply the transformation at different levels (Fig. 2).

We also implement a syntactic app obfuscation tool called `OCTOPUS` similar to DroidChameleon [44]. Specifically, `OCTOPUS` contains four levels of obfuscation: bytecode sequence obfuscation (*i.e.,* repacking, reassembling), identifier obfuscation (*i.e.,* renaming), call sequence obfuscation (*i.e.,* inserting junk code, call reordering, and call indirection) and encryption obfuscation (*i.e.,* string encryption). Then, we apply each level of obfuscation in `OCTOPUS` to each malware sample at a time, and perform test on the sample file (Sec. 7) after each obfuscation. If the test passes, we then apply the next obfuscation to the obfuscated sample. If the test fails, we restore the malware sample before the obfuscation. In our experiments, all semantic mutations including Random MRV and evolution/confusion attacks are performed after the syntactic obfuscation of `OCTOPUS`.

Malware detectors. We use a number of learning-based and signature-based malware detectors to evaluate the effectiveness of MRV. For learning-based malware detectors, we adopt AppContext [55] and Drebin [7]. **AppContext** leverages contextual features (*e.g.,* the events and conditions that cause the security-sensitive behaviors to occur) to identify malicious behaviors. In our experiments, AppContext generates around 400,000 behavior rows on our dataset (3852 apps), where each row is a 679-dimensional behavior vector. We conservatively label these behaviors (*i.e.,* marking a behavior as malicious only when the behavior is mentioned by existing malware diagnosis). The labeled behaviors are then used as training data to construct a Support Vector Machine (SVM) based classifier. **Drebin** uses eight features that reside either in the manifest file or in the disassembled code to capture the malware behaviors. Since

Table 2: Detection results of AppContext vs. Drebin on our dataset

Subjects	TP	FP	TN	FN
AppContext	1739/1917	107/1935	1828/1935	178/1917
Drebin	1879/1917	276/1935	1659/1935	38/1917

`TP. = True Positive, FP. = False Positive, TN. = True Negative, FN. = False Negative.`

Drebin is not open source, we develop our own version of Drebin according to its description [7]. Although Drebin extracts only eight features from a program, Drebin covers almost every possible combination of feature values resulting in a very large feature vector space. In fact, Drebin produces over 50,000 distinct feature values on our dataset (3852 apps). We perform ten-fold cross-validations to assess the effectiveness of AppContext and Drebin. Table 2 shows the performance of AppContext and Drebin on all subjects in our dataset.

For signature-based malware detectors, we leverage the existing anti-virus service provided by VirusTotal [6]. Specifically, we follow the evaluation conducted for Apposcopy [25] to pick the results of seven well-known anti-virus vendors (*i.e.,* AVG, Symantec, ESET, Dr. Web, Kaspersky, Trend Micro, and McAfee) and label an app as malicious if more than half of the seven suggest that the app is malicious. Following such procedure, only malware labeled as malicious are selected into our malware dataset, and thus all malware in our dataset can be detected by VirusTotal.

Producing mutated malware variants. Because many malicious servers of malware are blocked, causing malware to crash even before the repairing, we test the 1917 malware with 5,000 events randomly generated by Monkey and discard the crashed apps. This step gives us a set of 409 malware to perform program mutation. We then systematically apply `OCTOPUS`, evolution/confusion attacks, and random MRV to all 409 malware. In our experiments, the confusion attack assumes that the detectors using the SVM algorithm (which AppContext and Drebin use). To better analyze why MRV fails to produce evasive variants, for random MRV and evolution MRV, we produce both evasive and non-evasive variants and perform the detection after the variants are produced[12] (by skipping the detection step in Figure 2).

10 Result

10.1 Defeating existing anti-virus detection

Table 3 shows the malware variants generated through transformation of `OCTOPUS` , Random MRV, malware evolution attack and confusion attack, and the detection

[12]In fact, the variants are generated at each level, and one malware sample may result in multiple malware variants.

Table 3: Malware variants generated by different evasive techniques and undetected by VirusTotal

	O.	R.	E.	C.	F.
Transformable malware	409	121	314	58	341
Robust variants	1008	212	638	58	696
Undetected by VirusTotal	125	113	512	53	565

O. = OCTOPUS, R. = Random MRV, E. = Malware Evolution Attack, C. = Malware Confusion Attack, F. = Full Version of MRV

results of VirusTotal on the variants. We also show the result of the full version of MRV (the combination of confusion and evolution attack) in the last column (**F**). Therefore, the full version includes all malware variants produced by confusion and evolution attack. The row "Transformable malware" refers to the number of malware that can be mutated to a valid malware variant (*i.e.,* of all malware variants generated at different levels of an evasive technique, at least one of the malware variants can pass the testing). The row "Robust variants" shows the number of generated variants, and the last row shows the number of variants that can evade the detection of VirusTotal.

As shown in Table 3, full MRV can produce 565 evasive malware variants (to VirusTotal) while OCTOPUS and Random MRV can produce only 125 and 113 evasive malware variants, respectively. This result indicates that full MRV is much more effective in producing malware variants that can evade the detection of existing anti-virus software.

We investigate the malware variants produced by the full MRV that can still be detected by anti-virus software. We find that most of these variants contain extra payloads (*e.g.,* rootkit, another apk). The anti-virus software can detect them by identifying the extra payloads because our mutation transforms only the main program.

We notice that OCTOPUS is able to transform all 409 executable malware samples. The reason is that the bytecode sequence obfuscation (*i.e.,* repacking, reassembling) in OCTOPUS can be performed on all malware samples without crashing the malware. OCTOPUS produces more malware variants (1,008) than other evasive techniques because OCTOPUS includes four obfuscation levels while the other techniques have only three mutation levels. However, only one-eighth (125/1088) of the generated variants can evade VirusTotal's detection. The reason is that most of the variants are generated at the first two levels (*i.e.,* bytecode sequence and identifier obfuscation) and variants generated by more complicated obfuscation are less robust. For example, OCTOPUS can evade the detection by encrypting payloads. Some of the variants are, however, broken after data encryption, and thus fail the robustness test.

One result worth noticing is that confusion attack can successfully mutate only 58 malware into working malware variants. The reason is that confusion attack tries to mutate all contextual features at one time. Since the program transplantation technique is not perfectly robust (due to the fundamental limitation of static analysis), the likelihood for confusion attack to fail is high. The successfully mutated malware show strong evasion capability though. More than 90% of the variants (53/58) manage to evade the detection.

Table 4: Malware variants undetected (vs. all variants) by AppContext

Training Data	O.	R.	E.	C.	F.
All malware	0/1008 (0.0%)	2/212 (1.0%)	97/638 (15.2%)	56/58 (96.6%)	153/696 (22.0%)
All + 20% variants	0/806 (0.0%)	2/170 (1.2%)	89/510 (17.5%)	45/46 (97.8%)	134/556 (24.1%)
All + 40% variants	0/604 (0.0%)	1/127 (0.8%)	67/383 (17.5%)	34/35 (97.1%)	101/418 (24.2%)
All + 60% variants	0/403 (0.0%)	0/85 (0.0%)	43/255 (16.9%)	23/23 (100.%)	66/278 (23.8%)
All + 80% variants	0/302 (0.0%)	0/42 (0.0%)	24/128 (18.8%)	11/11 (100.%)	35/139 (25.2%)

O. = OCTOPUS, R. = Random MRV, E. = Malware Evolution Attack, C. = Malware Confusion Attack, F. = Full Version of MRV

10.2 Defeating learning-based detection

To assess the effectiveness of MRV in evading learning-based techniques, we not only use *all* original malware but also combine them with different portions of generated malware variants (20%, 40%, 60%, 80%) as the training data. We add a part of generated variants in the training dataset because in addition to measuring the effectiveness of MRV in evading the current detection model, we also want to test whether machine learning techniques can gradually detect latest generated variants by learning the evasive patterns of MRV from the existing variants.

Appcontext. Table 4 shows the detection results of AppContext. Each row represents the undetected variants vs. all variants by AppContext with different sets of training data. Clearly, full MRV is much more effective than OCTOPUS and the Random MRV and the percentages of variants being detected by AppContext are on the same level even with more variants in the training malware. The result indicates that the existing learning model fails to learn the evasive patterns of MRV.

It is expected to see none of the OCTOPUS variants can evade the detection of AppContext because the syntactic obfuscations in OCTOPUS would not affect the results of semantics-based detection such as AppContext. On the contrary, it is surprising to see that AppContext detects almost all variants produced by random MRV. Such result indicates that although random MRV is effective in befuddling the syntactic-based detection (*e.g.,* anti-virus software), it is not effective in evading semantics-based detection techniques.

It is also interesting to see that although evolution attack produces more well-functioning variants than confusion attack (Column "Mutated"), confusion attack is

Table 5: Malware variants undetected (vs. all variants) by Drebin

Training Data	O.	R.	E.	C.	F.
All malware	0/1008 (0.0%)	111/212 (52.4%)	460/638 (71.1%)	58/58 (100.%)	518/696 (74.4%)
All + 20% variants	0/806 (0.0%)	91/170 (53.5%)	352/510 (69.0%)	46/46 (100.%)	398/556 (71.6%)
All + 40% variants	0/604 (0.0%)	65/127 (51.2%)	260/383 (67.9%)	35/35 (100.%)	295/418 (70.6%)
All + 60% variants	0/403 (0.0%)	44/85 (51.8%)	170/255 (66.7%)	23/23 (100.%)	193/278 (69.4%)
All + 80% variants	0/302 (0.0%)	23/42 (54.8%)	76/128 (59.3%)	11/11 (100.%)	87/139 (62.6%)

`O. = OCTOPUS, R. = Random MRV, E. = Malware Evolution Attack, C. = Malware Confusion Attack, F. = Full Version of MRV`

Table 6: Details of Evolution Attack at each level (undetected vs. all)

Results	T.	L.	D.
Robust variants	178	316	144
Undetected by VirusTotal	77/178	296/316	139/144
Undetected by AppContext	21/178	15/316	61/144
Undetected by Drebin	73/178	272/316	115/144

T. = Temporal Features L. = Locale Features D. = Dependency Features

capable of making almost all produced variants evade the detection. The result indicates that AppContext is robust to detect certain unseen malware variants but has difficulty in differentiating certain types of known malware and benign apps.

Drebin. Table 5 shows the detection results of Drebin. Although originally Drebin detects more malware than AppContext (Table 2), Drebin performs worse on the full MRV dataset. Given different training malware, full MRV can consistently make over 60% testing variants get undetected by Drebin. One potential reason could be that AppContext leverages huge human efforts in labeling each security-sensitive behavior, while Drebin is a fully automatic approach, so overfitting is likely to occur in Drebin's model. However, we do notice that more MRV variants can evade the detection as the number of variants in training data increases. Such trend indicates that Drebin's model has potentials in detecting the evasive patterns of MRV variants.

We also notice that Random MRV becomes much more effective in evading Drebin than evading AppContext. The reason lies in the large number of syntactic features used in Drebin. The result indicates that mutating RTLD features can help to evade syntactic detection in general, regardless of using machine learning techniques.

10.3 Effectiveness of attacks at each level

For evolution attack, we also investigate the effectiveness of mutation at each RTLD levels. Table 6 shows the detailed detection results of evolution attack at each mutation level.

We have some interesting observations from Table 6. For example, for anti-virus software and Drebin, the level that produces the most number of evasive variants is on the locale feature level, while for AppContext, the level that produces the largest number of evasive variants is the dependency level. This result indicates that mutating at the locale feature level is more effective to the detectors using syntactic features (*e.g.,* VirusTotal, Drebin), while mutating at the dependency feature level is more effective for semantics-based detectors (*e.g.,* AppContext). Such result also indicates that the transformation sequence used in the evaluation (*i.e.,* temporal-locale-dependency) might not be the most optimal choice to evade some detectors. Ideally, we can explore different combinations of the mutation levels to maximize the number of undetected malware samples for each malware detector.

10.4 Evolution patterns of RTLD features

We summarize the evolution strategies for temporal features, locale features, and dependency features (Sec. 5). To maintain the maliciousness of mobile apps, we exclude the evolution patterns that change the malicious intentions (*i.e.,* aiming to exploit different resources/privileges).

Temporal features. We observe that lifecycle events are the temporal features that are most likely to evolve. In many malicious behaviors, the temporal feature evolves from a lifecycle event to a UI event, a system event or another lifecycle event. Moreover, such evolution can be intra-component (*i.e.,* the entrypoint is changed to another method in the same Android component) or inter-component (*i.e.,* the entrypoint is changed to a method in a different Android component). For inter-component evolution, the corresponding inter-component communications have been added to incorporate the evolution. Malware also frequently changes temporal features for malicious behaviors triggered by system events. Except some system events commonly observed in benign apps (*e.g.,* `android.intent.action.PACKAGE_ADDED`, `android.intent.action.BOOT_COMPLETED`), most temporal features such as system events have evolved to UI events or lifecycle events. The behaviors that were triggered by third-party ad-network intents (*e.g.,* `com.airpush.android.PushServiceStart`) also evolved[13].

Despite the frequent temporal-feature evolution, the evolution that changes UI events to other temporal features seldom happens. Based on the observation on overall evolution of several malware families, we find that the reason of such one-direction evolution is that to avoid

[13]Such evolution is due to that the emerging `Adblock` apps (*e.g.,* Airpush Detector [2], `Airpush` Opt-out [3]) force Adware authors to implement the malicious behaviors by themselves instead of leveraging third-party libraries such as `Airpush`.

being detected by users or dynamic app inspection, malware tend to decrease the frequency of triggering known malicious behaviors (by changing the triggering event to UI event) while triggering new attacks more frequently. Such evolution also confuses the static detection. Because UI events are also commonly used in benign apps, the temporal feature resulted from the evolution makes malware less identifiable for detection techniques.

Locale features. We observe that locale features often evolve with temporal features. The reason is that when malware authors change the entrypoint of the malicious behavior from one component to another, they tend to directly migrate the code to another component. Most of such evolution occurs for the malicious behaviors with fewer (*e.g.,* one or two) entrypoints and the security-sensitive methods are directly invoked in the entrypoint methods. We find that the most frequent changes of locale features are migration from an activity or receiver component to a service component. However, we observe less evolution the other way around (*i.e.,* evolution from service to activity or receiver). Such pattern is because the malicious behaviors in service components tend to be continuous (*e.g.,* downloading, monitoring) while activity or receiver components cannot support such continuity (unless starting another thread). So when entrypoints for the malicious behaviors change to other components, the malware authors still keep malicious behaviors in the service components and direct the execution from other components through inter-component communications.

Dependency features. The most common evolution for dependency features is the change of the controls through inter-component communications. Such evolution can confuse the malware detection techniques since the generated PDG can reflect the control dependencies only between malicious behavior and intent values. Connecting the control dependencies between the malicious behavior and the original value requires precise analysis of inter-component communication. However, the use of such analysis is absent or limited in existing malware detectors. The other frequent evolution is based on attributes of external entities (including `Internet connection` and `telephony manager`) that aim to update the malicious logic of the malware. The Internet connection reflects the command & control behaviors through network servers. The package manager and telephony manager suggest that malware controls the malicious behavior based on the installed apps on the phone and the IMEI number or network server of the phone. Such evolution does not aim to evade detection, but to update the malicious logic of the malware.

Composite Evolution. We have observed several interesting cases that combine the changes of all three types of features. One representative case is a malicious app trying to obtain `NetworkInfo` on the phone to launch malicious behaviors based on different types of network connection. The original malware leverages the system event `android.net.conn.CONNECTIVITY_CHANGE` to get notified when network connection changes so it can obtain the `NetworkInfo` from the system Intent message for new network connections. As the system event may be used as temporal features, the app evolves to leverage the system event `android.intent.action.USER_PRESENT` . Such event will get malware notified when a user is present after unlocking the screen. Then the malware starts a service and uses a timer in the service to repetitively invoke `getNetworkInfo` every 20 minutes. In this way, the malware changes its original temporal feature value (`CONNECTIVITY_CHANGE`) to a value (`USER_PRESENT`) completely unrelated to network connection. Meanwhile, it changes its locale feature value from `receiver` to `service`, and adds the control dependency with timer. MRV can be especially powerful in detecting such sophisticated evolution.

11 Discussions

Limitation of underlying infrastructure Our framework suffers from the limitations of standard static analysis in handling threading, alias, virtual methods, reflection, callbacks, and native). Also due to the limitation of expressiveness of Jimple, the intermediate representation (IR) for program analysis tasks used in Soot in transplantation framework, the conversion from Dalvik code to Jimple may not preserve the complete program behavior. This could be addressed by introducing other tools such as dex2jar for program transformation.

Limitation of App testing In testing processes for transformed apps, the testing only checks two things: whether the program crashes or not and whether the malicious statements get invoked or not. Due to the unavailability of systematic functional test cases in regression tests, the testing cannot cover all the cases. This could be improved by manually creating regression tests for the malicious functionality. Additionally, it is infeasible to test whether malware become less profitable or easier to be detected without users' support.

Adware Threat in experiments In experiments, we view one family of adware (*i.e.,* AdsDisplay) as one malware family due to the fact that both malware and adware are "unwanted software (UNP)", and distinct significantly from benign apps according to the recent study [46].

Threat to machine learning based classifier In practice, any machine learning based system always has parameters and thresholds. In our attack model, we did not discuss too much about the determination of thresholds for decisions. We assume the parameters are already searched for the optimal values via cross validation and prior knowledge.

Mitigation of MRV The possibility of MRV largely depends on the effectiveness of malware features used in detection techniques to capture the tendency of the possible malware variation. Most existing malware detection techniques summarize the patterns based on static and behavior analysis of malware. Summarizing pattern without revealing the evolving tendency might not be sufficient to fight MRV. The evolving pattern of malware can be deemed as a concept drift in learning problems. Concept drift [42] is used as a generic term to describe the relation between the input and the target variable of a learning problem changes over time. Adversary activities can be one of the major source to cause the change [60]. One way to cope with MRV is to adopt drift-aware adaptive learning algorithm [18,24,43] or concept drift detection [41] in the malware detection.

Coping the malware evolution can also be deemed as a long-term dependency learning problem [14]. Not only we need to investigate recent malware, but also to study the gradient changes in a long period. Long Short Term Memory networks (LSTMs) [30] are a special kind of recurrent neural network which is designed for learning long-term dependencies. It will be an interesting direction to apply LSTMs on malware detection.

In addition to concept drift adaptation and neural networks, malware trend analysis [53] based on temporal patterns [33,48] of malware behaviors is also highly desirable in assisting machine learning models to detect the evolving code structures and behaviors.

12 Related Work

Our work is mainly related to the following works.

Evasive Techniques Metamorphic malware [38], polymorphic malware [36] and other obfuscation techniques [22] have been developed to evade malware detection [57]. Semantic signature [21], behavior graphs [27] and other semantics aware techniques [50], [55], [9] have been developed to defeat against malware. To study how anti-malware products are resistant against transformed mobile apps, Droidchameleon [44] is developed as a systematic framework with various transformation techniques for mobile app study and they found the transformed apps can easily evade detection. Replacement attack [51] is proposed to poison behavior-based specifications by concealing similar behaviors of malware variants. Replacement attack can impede malware clustering [13]. MalGene [34] is developed to automatically locate evasive behavior in system call sequences and therefore extract evasion signatures. Different from these works, MRV can evade both anti-virus tools and machine learning based classifiers.

Recent evaluations [46] on ML-based malware detection techniques suggest that more feature number does not necessarily improve the performances due to the non-informative features [7] and noisy features. Recent study on PDFrate [45] evaluates existing learning-based malware detection techniques in different evasion scenarios. Xu et al. [52] propose an evading technique based on classification score feedback that can manipulate PDF malware samples to evade detection of PDFrate and Hidost. Carmony et al. [17] manipulate JavaScript payload in PDF malware to evade detections. The major difference between these works and MRV is that MRV requires much less knowledge about machine learning model to launch the attack. Moreover, prior work mainly focuses on pdf malware while MRV automatically generates malware variants for android apps.

Adversarial Machine Learning Adversarial machine learning [15, 16, 32, 45] studies the effectiveness of machine learning techniques against an adversarial opponent. Adversaries frequently attempt to break many of the assumptions that practitioners make (*e.g.,* data has various weak stochastic properties; data do not follow a stationary data distribution). Generally, the adversary is assumed to have full (or partial) information related to three components of learner: learning algorithm, feature space and training dataset. In our work, we propose a targeted attack assuming a malware developer knows the information about the feature space and classifier.

Program Transplantation We leverage program transplantation technique for program transformation. Given an entry point in the donor and a target implantation point in the host program, the goal of automated program transplantation [10,39,47] is to identify and extract an organ, all code associated with the feature of interest, then transform it to be compatible with the name space and context of its target site in the host. In this work, we broaden program transplantation concept to intra-app transplantation that needs to deal with the side-effect of the removal of the organ (compared to inter-app transplantation), where effective heuristics are proposed to automatically identify the target implantation point in the host program.

Contextual-based malware detection To identify potential malware among rapidly increasing number of published apps, existing researches leverage both dynamic and static analysis. Pegasus [19] constructs permission event graphs using static analysis to model the effects of the event system and API semantics, and performs model checking to enforce the policies specified by users. AppIntent [56] presents a sequence of GUI events that lead to data transmissions and lets analysts decide whether the data transmissions are intended. AsDroid [31] detects stealthy app behaviors by identifying mismatches between API invocations and the text displayed in the GUIs. DroidSIFT [58] builds the weighted contextual API dependency graph to fight against malware, variants and zero-day attacks. MassVet [20] determines whether a new app is malicious or not by finding the commonalities between the new one and the known malware based on features generated from UI structure and method control-flow graphs. Triggerscope [26] seeks to automatically identify logic bombs in Android malware using symbolic execution, path predicate reconstruction and minimization, and interprocedural control-dependency analysis to enable the precise detection and characterization of detecting triggers. BareCloud [35] is developed to detect evasive malware based on bare-metal dynamic malware analysis by comparing malware behavior observed in the transparent bare-metal system with other popular malware analysis systems including emulation-based and virtualization-based analysis systems. Because all these works leverage contextual/non-essential features the RTLD feature model can be used to generate malware variants to evade the detection of these techniques.

13 Conclusion

In this paper, we propose two attacks that mutate malware variants to evade detection. The core idea is changing the behaviors that the malware detector concerns. To achieve this goal, we leverage static analysis, phylogenetic analysis, machine learning, and program transplantation techniques to systematically produce the new malware mutations. To the best of our knowledge, it is the first effort towards solving malware evasion problem using malware feature confusions and evolutions without *any* knowledge of the underlying detection models. MRV opens intriguing new problems. First, proposed attacks can be used to evaluate the robustness of the malware detectors and therefore quantify the differentiability of particular fe1atures. Second, MRV can help discover the potential attack surface so MRV can be further used to iterative design of malware detectors. Finally, the program transplantation framework capable of changing malware features and behaviors can be written as malicious payload that resides in malware itself so as to create adaptive malware.

References

[1]

[2] Airpush Detector. https://goo.gl/QVn82.

[3] Airpush Opt-out. http://www.airpush.com/optout/.

[4] Contagio mobile - mobile marewale mini dump. http://contagiominidump.blogspot.com/.

[5] Virusshare.com. http://virusshare.com/.

[6] Virustotal - free online malware, virus and URL scanner. https://www.virustotal.com/.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[8] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the android permission specification. In *Proc. CCS*, 2012.

[9] D. Babic, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *CAV*, 2011.

[10] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *ISSTA*, 2015.

[11] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *SOAP*, 2012.

[12] A. D. Baxevanis and B. F. F. Ouellette. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. John Willey and Sons, 2005.

[13] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[14] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[15] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *ECML*. 2013.

[16] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In *29th ICML*, 2012.

[17] C. Carmony, M. Zhang, X. Hu, A. V. Bhaskar, and H. Yin. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*, 2016.

[18] D. Charles, A. Kerr, M. McNeill, M. McAlister, M. Black, J. Kcklich, A. Moore, and K. Stringer. Player-centred game design: Player modelling and adaptive digital games. In *Proceedings of the Digital Games Research Conference*, volume 285, page 00100, 2005.

[19] K. Chen, N. Johnson, V. D'Silva, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song. Contextual policy enforcement for android applications with permission event graphs. In *NDSS*, 2013.

[20] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, 2015.

[21] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE SP*, 2005.

[22] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. In *Technical Report of University of Auckland*, 1997.

[23] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security*, 49.

[24] F. Fdez-Riverola, E. L. Iglesias, F. Díaz, J. R. Méndez, and J. M. Corchado. Applying lazy learning algorithms to tackle concept drift in spam filtering. *Expert Systems with Applications*, 33(1):36–48, 2007.

[25] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*, 2014.

[26] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android apps. In *IEEE SP*, 2016.

[27] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE SP*, 2010.

[28] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android application. In *PLDI*, 2014.

[29] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Mobisys*, 2014.

[30] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[31] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *ICSE*, 2014.

[32] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *AISec'11*, 2011.

[33] C. M. Hurvich and C.-L. Tsai. Regression and time series model selection in small samples. *Biometrika*, 76(2).

[34] D. Kirat and G. Vigna. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *CCS*, 2015.

[35] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security*, 2014.

[36] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2006.

[37] H. W. Kuhn and B. Yaw. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955.

[38] F. Leder, B. Steinbock, and P. Martini. Classification and detection of metamorphic malware using value set analysis. In *Malware*, 2009.

[39] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE*, 2015.

[40] Monkey test generator home. http://developer.android.com/tools/help/monkey.html.

[41] K. Nishida and K. Yamauchi. Detecting concept drift using statistical testing. In *International conference on discovery science*, pages 264–269. Springer, 2007.

[42] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks*, 51(12):3448–3470, 2007.

[43] M. Pechenizkiy. Predictive analytics on evolving data streams anticipating and adapting to changes in known and unknown contexts. In *High Performance Computing & Simulation (HPCS), 2015 International Conference on*, pages 658–659. IEEE, 2015.

[44] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *ASIACCS '13*, 2013.

[45] N. Rndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *IEEE SP*, 2014.

[46] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *ACSAC*, 2015.

[47] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *PLDI*, 2015.

[48] K. Tamura, D. Peterson, N. Peterson, G. Stecher, M. Nei, and S. Kumar. Mega5: molecular evolutionary genetics analysis using maximum likelihood, evolutionary distance, and maximum parsimony methods. *Molecular biology and evolution*, 2011.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASON*, 1999.

[50] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. Airbag: Boosting smartphone resistance to malware infection. In *NDSS*, 2014.

[51] Z. Xin, H. Chen, X. Wang, P. Liu, S. Zhu, B. Mao, and L. Xie. Replacement attacks on behavior based software birthmark. In *ISC*. 2011.

[52] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *NDSS*, 2016.

[53] J.-M. Yang, W.-C. Liao, W.-C. Wu, and C.-Y. Yin. Trend analysis of machine learning-a text mining and document clustering methodology. In *New Trends in Information and Service Science*, 2009.

[54] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *FASE*. 2013.

[55] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *ICSE*, 2015.

[56] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, 2013.

[57] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[58] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS*, 2014.

[59] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE SP*, 2012.

[60] I. Zliobaite, M. Pechenizky, and J. Gama. An overview of concept drift applications. *Big Data Analysis: New Algorithms for a New Society. SBD*, 16:91–114.