Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps

Wei Yang
University of Illinois at Urbana-Champaign
weiyang3@illinois.edu

Deguang Kong
Yahoo Research
doogkong@gmail.com

Tao Xie
University of Illinois at Urbana-Champaign
taoxie@illinois.edu

Carl A. Gunter
University of Illinois at Urbana-Champaign
cgunter@illinois.edu

ABSTRACT

Existing techniques on adversarial malware generation employ feature mutations based on feature vectors extracted from malware. However, most (if not all) of these techniques suffer from a common limitation: feasibility of these attacks is unknown. The synthesized mutations may break the inherent constraints posed by code structures of the malware, causing either crashes or malfunctioning of malicious payloads. To address the limitation, we present Malware Recomposition Variation (MRV), an approach that conducts semantic analysis of existing malware to systematically construct new malware variants for malware detectors to test and strengthen their detection signatures/models. In particular, we use two variation strategies (*i.e.,* malware evolution attack and malware confusion attack) following structures of existing malware to enhance feasibility of the attacks. Upon the given malware, we conduct semantic-feature mutation analysis and phylogenetic analysis to synthesize mutation strategies. Based on these strategies, we perform program transplantation to automatically mutate malware bytecode to generate new malware variants. We evaluate our MRV approach on actual malware variants, and our empirical evaluation on 1,935 Android benign apps and 1,917 malware shows that MRV produces malware variants that can have high likelihood to evade detection while still retaining their malicious behaviors. We also propose and evaluate three defense mechanisms to counter MRV.

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation;**

KEYWORDS

Malware detection, Adversarial classification

1 INTRODUCTION

Along with the exponential growth of markets of mobile applications (*apps* in short) comes the frequent occurrence of mobile malware. According to the McAfee Security Report [27], more than 37 million mobile malware samples were detected over the six months preceding the report-writing time. To fight against malware, a signature-based technique extracts malicious behaviors as signatures (such as bytecode or regular expression) while a more complicated machine-learning-based technique [23] learns discriminant features from analyzing semantics of malware.

To defeat these detection techniques, malware authors constantly produce new variants of existing malware families. Recent studies [33, 41, 34] show that performance of both signature-based and learning-based malware detection techniques can be degraded by carefully-crafted malware variants. To increase the robustness of malware detectors against malware variants, we need to be proactive and take potential adversarial scenarios into account in designing malware detectors. To enable the proactive design of malware detectors, existing work [16, 11, 19, 13] has been proposed to evaluate malware detectors in adversarial settings. Such work envisions potential attack scenarios and manipulates (adds, changes, or removes) features[1] extracted from malware according to the envisioned attacks. The malware detectors are then used to identify malware variants through the manipulated features. The performance of the malware detection on manipulated features is expected to be lower than the original detection. The robustness of the malware detectors are further evaluated based on performance degradation.

However, the validity of these evaluations is questionable due to impracticality of the attacks. The attacks used in the prior work manipulate the feature vectors of a malware sample without considering feasibility and impact of the mutation. In other words, when applying the changes made in feature vectors to the malware's code, the changes may cause the malware to crash, cause undesired behaviors, or disable malicious functionalities (sometimes the modified code cannot even be compiled).

There are three practical constraints to craft a realistic attack:

Preserving Malicious Behaviors. The mutated malware should maintain the original malicious purposes, and therefore simply converting the malware's feature values to another app's feature values is likely to break the maliciousness. For example, malicious behaviors are usually designed to be triggered under certain contexts (to avoid user attention and gain maximum profits [43]), and the controlling logic of the malware is too sophisticated (*e.g.,* via logic bombs and specific events) to be changed.

[1]Malware detectors use feature sets for signatures or detection models.

978-1-4503-5345-8/17/12…$15.00
DOI: 10.1145/3134600.3134642

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 cause the app to crash at runtime.

Evading Malware Detectors. To evade a malware-detection model, an adversary needs to identify the features and compute the feature values that can evade detection without breaking the malware. Doing so usually requires an adversary to possess internal knowledge and understanding of the 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 single malware-detection model is too specific to successfully produce evasive variants, especially if the malware detector (*e.g.,* VirusTotal [5]) is based on combining multiple models or techniques.

To create an attack satisfying these three constraints, we employ Malware Recomposition Variation (MRV) consisting of two mutation strategies, *Malware Evolution Attack* and *Malware Confusion Attack* (Section 3). The advantage of our mutation strategies is that the strategies can produce high percentages of feasible feature mutations (suggested in our evaluation), thus greatly enhancing the feasibility of the attacks. The insight is that feature mutations are less likely to break the apps when the mutations follow feature patterns[2] of existing malware. To mutate app features in black-box scenarios, we create a substitute model named as RTLD (Section 2) approximating the models of malware detectors. Such methodology has been widely used to launch successful black-box attacks in prior work [21, 30, 26]. RTLD generally reflects the susceptibility of a detection technique to the mutations of malware feature values.

To apply the mutation strategies without breaking dependencies and functionalities in an app, we develop a new technique, inspired by program transplantation [36], to reuse the existing implementations 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 (transplanting a feature in one app/component/method, *i.e.,* donor, to a different app/component/method, *i.e.,* host). By leveraging the existing implementations, this technique enables systematic and automatic mutations on malware samples while aiming to produce well-functioning apps.

We also propose and evaluate three defense mechanisms to strengthen the robustness of malware detectors against MRV attacks. (i) *Adversarial Training*. Training a new model with the combination of the generated malware variants and original training data. (ii)*Variant Detector*. Developing a detector in addition to the original malware detector to detect whether an app is a variant derived from existing malware. (iii)*Weight Bounding*. Bounding the feature weights in the original malware detector to make feature weights more evenly distributed[3].

Main Contributions. This paper makes the following main contributions.

[2]In our mutation strategies, the feature patterns are extracted from malware evolution histories and existing evasive malware.

[3]Such defense mechanism forces the increasing number of mutations to craft adversarial samples; therefore, the attack will likely become infeasible (our empirical result suggests that the number of working adversarial samples dramatically decreases as the number of mutations increases).

- ***Attacks.*** We propose two practical attacks (*feature evolution attack* and *feature confusion attack*) to effectively mutate existing malware for evading detection (Section 3).
- ***Observation.*** We evaluate the robustness of detection models and the differentiability of selected features of malware detectors by systematically and automatically applying proposed attacks to existing malware detectors (Section 7).
- ***Characterization.*** We propose an RTLD feature model that characterizes and differentiates contextual and essential features of malicious behaviors (Section 2).
- ***Framework.*** We develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation to automatically mutate app features (Section 4).

Related Work. Smutz et al. [38] propose mutual agreement analysis to detect classifier evasions in malware detectors. To evaluate the improvement over the Drebin malware detector, Smutz et al. withhold some malware families from the training set and use malware samples from these families to check the performance of malware detectors. Our work differs from their work in two aspects. First, their approach works for only ensemble classifiers, while our approach works for any type of malware detectors. Second, they use existing malware samples to mimic the attack of unknown malware families, while we generate previously unknown malware variants to test the robustness of malware detectors.

Grosse et al. [19] investigate how adversarial perturbation would affect malware detectors based on deep neural networks. Grosse et al. mutate malware features based on the forward derivative [31] of the neural network to evade detection. Grosse et al. do not apply the computed feature mutations to the malware bytecode. Instead, they choose a conservative mutation strategy by adding only manifest features (*i.e.,* features extracted from the manifest file, `AndroidManifest.xml`). In comparison, our proposed attacks are more comprehensive (adding, removing, or changing features) and more practical (changing both the manifest file and dex code).

Demontis et al. [16] investigate performance of malware detectors by applying a few previous attacks [12, 22]. Demontis et al. propose to perform feature mutations and obfuscations to evade malware detection. However, their work does not apply feature mutations on malware bytecode. It is possible that the computed feature mutations are infeasible attacks. Hu et al. [21] propose to leverage generative adversarial network [18] to generate adversarial malware samples. They also do not apply feature mutations on malware bytecode, thus leading potential infeasible attacks.

2 RTLD FEATURE MODEL

We characterize semantic features of mobile apps using the RTLD[4] feature model, which aims to reflect the essential malicious behaviors while balancing between the computational efficiency and accuracy. The RTLD feature model is a general model summarizing the essential features (*i.e.,* security-sensitive resources) and contextual features (*e.g.,* when, where, how the security-sensitive resources are obtained and used) commonly used in malware detection.

The RTLD features cover four main aspects: *Resource* (what security-sensitive resources malicious behaviors obtain), *Temporal* (when the malicious behaviors are triggered), *Locale* (where the

[4]RTLD is short for Resource, Temporal, Locale, and Dependency.

malicious behaviors occur), and *Dependency* (how the malicious behaviors are controlled).

The advantages of using the RTLD features are two-fold. First, we can learn a substitute model approximating the targeted detector using the RTLD features. Based on the transferability property [29, 26, 39], some adversarial samples generated based on the substitute model can also evade the original detectors.

The second advantage of the RTLD features is the separation of essential features and contextual features. To form an informative feature set for signature or detection model, existing malware detection tends to include as many features as possible. For example, Drebin, a recently published approach of malware detection [7], uses the feature set containing 545,334 features. A recent study [35] shows that such large feature set has numerous non-informative or even misleading features. Two of our observations confirm this finding. (i) Malware detectors often confuse non-essential features in code clones as discriminative features. Copy-paste practice is prevalent in the malware industry, resulting in many code clones in malware samples [15]. Because the same code has appeared in many malware instances, learning-based detectors may regard non-essential features (*e.g.,* minor implementation detail) in code clones as major discriminant factors (because the same pieces of code appear in many malware samples but not in benign apps). (ii) Using a universal set of features for all malware families would result in a large number of non-essential features to characterize each malware family, because the features essential to malicious behaviors are different for each family. The separation of essential features and contextual features in the RTLD model enables our generated attacks to pinpoint the features that are not critical to the malicious behaviors but confusing to the classifiers.

We use the simplified code snippet of the DougaLeaker malware[5] shown in Figure 1 to illustrate the feature model. The code snippet shows two malicious behaviors of the DougaLeaker malware. First, the code snippet saves the Android ID and telephone number of the victim device to global class `User` when the app starts (Lines 5-7 in Figure 1b). Then the code snippet reads contacts on the victim device (Lines 8-13 in Figure 1b) and sends the contacts to a malicious server (Line 20 in Figure 1b). The code snippet also starts a service that sends the Android ID and telephone number to the malicious server through text messages between 11PM and 5AM.

The **resource** features describe the security-sensitive resources exploited by malicious behaviors while the dependency features further represent how the malicious behaviors are controlled. We locate resource features by constructing call graphs and identifying call-graph nodes of the security-sensitive methods (including methods for accessing permission-protected resources and methods for executing external binaries/commands). We compile the list of security-sensitive methods based on PScout [8] and construct the call graphs using the SPARK call-graph algorithm implemented in Soot [40]. The call graphs represent the invocation relationships between the app's entrypoints and permission invocations. We save the entrypoints of the call graphs in this step to trace back to the other features in later steps. For the DougaLeaker example, we can locate the `HttpPost` method invocation (not shown in Figure 1) in `exec_post` along with the `sendTextMessage` method invocation

[5]MD5 of the malware is e65abc856458f0c8b34308b9358884512f28bea31fc6e326f6c1078058c05fb9.

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

(a) `User` class of DougaLeaker malware

```
1  public class MainActivity extends Activity{
2  public void onCreate(android.os.Bundle b){
3  super.onCreate(b);
4  this.requestWindowFeature(1);
5  User u = (User) getApplication();
6  u.androidid = Settings.Secure.getString(getContentResolver(),
       "android_id");
7  u.tel = getSystemService("phone").getLine1Number();
8  if(isRegisterd(u.androidid)){
9  Cursor cursor = managedQuery(ContactsContract.Contacts.
       CONTENT_URI, 0, 0, 0, 0);
10 while (cursor.moveToNext() != 0) {
11 this.id = cursor.getString(cursor.getColumnIndex("_id"));
12 this.name = cursor.getString(cursor.getColumnIndex("
       display_name"));
13 this.data = new StringBuilder(String.valueOf(this.data)).
       append("name:").append(this.name).toString();
14 }
15 cursor.close();
16 }else{
17 startService(new Intent(getBaseContext(), MyService.class));
18 }
19 }
20 this.exec_post(this.data);}} //sending contacts through
       HttpPost
```

(b) `MainActivity` of DougaLeaker malware

```
1  public class MyService extends Service{
2  public int onStartCommand(Intent intent, int flags, int startId
       ){
3  User u = (User) getApplication();
4  String text = "android_id = " + u.androidid + "; tel =" + u.tel;
5  Date date = new Date();
6  if(date.getHours>23 || date.getHours< 5 ){
7  android.telephony.SmsManager.getDefault().
       sendTextMessage(this.number, null, text, null, null);
8  }
9  return;}}
```

(c) `MyService` of DougaLeaker malware

Figure 1: Motivating Example: DougaLeaker malware

(Line 7 in Figure 1c) in `onStartCommand` in the call graph. Due to space limit, we omit many details here.

The **temporal** features describe the contexts when the malicious behaviors are triggered. To extract the temporal features, we identify three categories of temporal features based on the attributes of their entrypoints. (i) For system events handled by intent filters, their entrypoints are lifecycle methods. The components of the lifecycle methods should have intent filters specified. (ii) For both system events captured by event-handling methods and UI events, their entrypoints should be event-handling methods. (iii) For lifecycle events, their entrypoints are lifecycle methods, and

these lifecycle methods have not been invoked by other events (due to inter-component communications).

The **locale** features describe the program locations where the malicious behaviors occur. The location of a malicious behavior is either an Android component (*i.e.,* `Service`, `Activity`, and `Broadcast Receiver`) or concurrency constructs (*e.g.,* `AsyncTask` and `Handler`). Malicious behaviors get executed when these components are activated. Due to the inter-component communication (ICC) in an Android app, 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 of a task (*i.e.,* whether the execution of the task is in the foreground or background) and continuity (*i.e.,* whether the task is once-off execution or a continuous execution, even after exiting the app). For example, if a permission is used in a Service component (that has not been terminated by `stopService`), the permission use is running in the background, and it is also a continuous task (even after exiting the app).

The **dependency** features describe the control dependencies of invocations of malicious behaviors. A control dependency between two statements exists if the truth value of the first statement controls whether the second statement gets executed. Malware frequently leverage external events or attributes to control malicious behaviors. For example, a `DroidDream` malware sample leverages the current system time to control the execution of its malicious payload. It suppresses its malicious payload during the day but allows the payload's executions at late night when users are likely sleeping.

We construct inter-procedure control-flow graph (ICFG) to extract the dependency features. Based on the ICFG, we construct the subgraphs from each entrypoint to a resource feature (*i.e.,* security-sensitive method call). For each subgraph, we traverse the subgraph to identify the conditional statements that the security-sensitive method invocation is control-dependent on. The value of a conditional statement is used to decide which program branch to take in runtime executions, and thus decide whether a security-sensitive method invocation on one of the program branches can be executed or not. We say that such conditional statement controls the invocation of the method. Finally, we save the set of extracted conditional statements as dependency features with the resource features and the corresponding location/temporal features. Figure 8 shows the ICFG of the `onCreate` and `onStartCommand` methods. As shown in the figure, the `sendTextMessage` method on Line 7 in `onStartCommand` (Figure 1c) is controlled by the conditional statement on Line 6 in `onStartCommand` and the conditional statement on Line 8 in `onCreate` (Figure 1b). On the other hand, the `exec_post` method in `onCreate` is not controlled by any conditional statement, and thus the security-sensitive behavior in `exec_post` does not have any dependency feature.

3 MUTATION STRATEGY SYNTHESIS

In this section, we present our techniques of synthesizing strategies to mutate app features in black-box scenarios. To address the challenge that adversaries have no knowledge about malware detection techniques (e.g., features, models, algorithms) in black-box scenarios, we develop two attacks: evolution attack and confusion attack.

3.1 Evolution Attack

In evolution attack, instead of developing targeted malware to evade specific detection techniques, we come up with a more general defeating mechanism: *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 mutation strategies, if automated, can be systematically employed on a large set of malware samples, enabling the exploitation to identify much more blind spots of existing detection.

The main insight for malware evolution attack is that malware creation is similar to biological evolution process (*i.e.,* copy and edit of the patterns). To interpret how malware evolve and understand the differences among variants of a malware family, we conduct a phylogenetic analysis for each family of malware[6]. We believe that capturing the subtle differences among evolving malware patterns can help *mimick new malware*.

Phylogenetic analysis on each family of malware. A phylogenetic evolutionary tree [10] is a branching diagram (*a.k.a* evolution tree) that shows the inferred evolution relations among different samples based on the similarities and differences in their feature representations. In the context of malware phylogenetic analysis, we aim to understand how each family of malware evolves.

We perform our study using the aforementioned RTLD feature type due to its competitive performance for classifying malicious apps. We conduct malware phylogenetic analysis from searching the *common shared* feature set and *divergent feature* set. The pairwise distance between malware samples is defined based on the ratio of the number of features in the common shared feature set to the number of features in the total feature set (including the divergent feature set).

Pairwise distance computation. Note that in the RTLD feature type, each mobile app actually corresponds to multiple lines of feature vectors. Therefore, we align the feature vectors from any two mobile apps, using the (permission, API) key. In other words, if the (Permission, API) values of two feature vectors are the same, we further study similarities between the two feature vectors. Otherwise, we stop comparing the two feature vectors since comparing two feature vectors with different API methods is not worthwhile. Let p_i be the API set carried by app i, p_j be the API set carried by app j, and then the similarity between apps i and j is defined as:

$$S_{ij} = \frac{p_i \cap p_j}{p_i \cup p_j}, \tag{1}$$

where $\cap$ is the intersection operation of the two sets and $\cup$ is the union operator of the two sets. We say that Eq.(1) captures the *coarse-grained* difference between two apps since it considers only the API method information.

[6]We investigate malware evolution by family due to the fact that most of new malware variants come from within existing families. A recent study [37] suggests that the number of malware families has remained relatively constant over the years whereas the number of variants within a family has been growing rapidly.

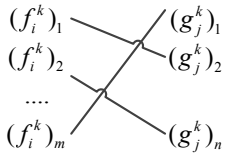

Figure 2: Illustration of fine-grained matching of feature vectors $[(f_i^k)_1, (f_i^k)_2, ..., (f_i^k)_m]$ and $[(g_j^k)_1, (g_j^k)_2, ..., (g_j^k)_n]$ regarding API method k from two apps i and j.

If S_{ij} is far below a threshold, this fact suggests that the two apps share very few similar behaviors defined by API methods. Therefore, we stop fine-grained comparisons of other feature vectors due to the large behavior gap.

If S_{ij} is above a certain threshold, this fact indicates that the two apps share many common behaviors defined by API methods. Then we study each API method in a finer-grained way, and check how the two apps are aligned regarding each API method. This technique invokes a more thorough treatment and diversity comparisons of feature vectors abstracted from both apps. We name such technique as *fine-grained* app behavior analysis.

In the *fine-grained* app behavior analysis, for the API method k, let the corresponding m feature vectors involving API method k in app i be $(f_i^k)_1, (f_i^k)_2, ..., (f_i^k)_m$, and the corresponding n feature vectors involving API method k in app j be $(g_j^k)_1, (g_j^k)_2, ..., (g_j^k)_n$, where feature vectors $(f_i^k)_\ell$ and $(g_j^k)_\ell$ are both d-dimensional feature vectors. We need to find the best alignment of two groups of feature vectors with respect to API method k. Note that in fact this problem can be abstracted as a maximum matching problem in a bipartite graph, where the first disjoint set is $[(f_i^k)_1, (f_i^k)_2, ..., (f_i^k)_m]$, and the second disjoint set is $[(g_j^k)_1, (g_j^k)_2, ..., (g_j^k)_n]$, and the edges between the two sets find the maximum matching. Figure 2 illustrates the feature vector matching process between two apps. Note that each bit in the feature vector $(f_i^k)_r$ is a binary value, where 1 denotes the existence of this feature. Therefore, for each pair of feature vectors, we define its distance W as the number of equivalent bits vs. the number of all feature bits, *i.e.,*

$$W((f_i^k)_r, (g_j^k)_t) = \frac{\#\text{shared feature bits by} (f_i^k)_r \text{and} (g_j^k)_t}{\#\text{feature vector length}}. \quad (2)$$

Given the pairwise feature distance, we can refer to a Hungarian-type Algorithm [24] to find the best matching of two groups of feature vectors and also get the subtle difference regarding each API method. To this end, we have obtained all pairwise distances among different apps via coarse-grained and fine-grained app analyses.

Generation of phylogentic tree. We then feed the pairwise distances of mobile apps to the phylogenetic tree generation algorithm, namely, Unweighted Pair Group Method Average algorithm (UPGMA) [25]. The UPGMA algorithm is a simple bottom-up hierarchical clustering algorithm that is most popularly used for creating phenetic trees. UPGMA builds a rooted tree (*a.k.a* dendrogram) that reflects the structure present in the similarity matrix. In each step, the nearest two clusters are merged into the higher-level cluster by averaging all pairwise sample distances. Please refer to the appendix of this paper for examples of phylogentic tree for the `AdDisplay` family and `Droid_Kunfu` family.

There are two key insights obtained from the phylogentic analysis:

- Even for the malware samples in the same sub-family, their distances may not be as small as those from different sub-families if only API methods are considered as "*features*".
- There are many subtle differences that cannot be captured by API methods alone, such as those differences in UI types and entry points, which actually provide complementary sources to understand how malware evolve.

Generation of candidate feature mutation set. Although phylogentic analysis provides similarities between different malware samples, in practice, we are more interested in knowing how each feature type evolves in the same family because the evolution analysis provides much information about the *feasibility* and *statistical* frequency regarding different feature types.

- Feasibility (F): as long as this feature value corresponding to a particular feature type has been mutated, we view it as *feasible*. In particular, if mutation of the i-th type feature exists, then $F(i) = 1$; otherwise, $F(i) = 0$ [7].
- Statistical Frequency (Sf): we count the number of feature mutations given an API method. In particular, if mutation of the i-th type feature appears for n times, then $Sf(i) = n$; otherwise, $Sf(i) = 0$.

Then we rank different feature mutations corresponding to each API method in a principled way, *i.e.,* making a balance between the feasibility and statistical frequency. Let the ranking function $R(.)$ be

$$R(i) = \alpha F(i) + (1 - \alpha) Sf(i); \quad (3)$$

where $\alpha = 0.1$ in our setting, and i denotes mutation of the i-th type feature. We sort $R(.)$ in the descending order, and select the top x features for mutations[8] as our candidate feature-mutation type, and feed them to the program mutation engine for generating new malware mutants.

3.2 Confusion Attack

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 complements the Malware Evolution Attack because learning-based malware detectors can be robust to malware evolution [37]. Such defeating mechanism is based on the observation that malware detectors (based on a classifier) could be easily misled by feature omissions and substitutions. In confusion attack, the main idea is to mimic the malware that can evade detection, *i.e.,* confusing the malware detectors by modifying the feature values that can be shared by malware samples and benign apps. In particular, we mutate differentiating feature values (*i.e.,* feature values that exist in *only* malware samples) to confusing feature values (*i.e.,* feature values that exist in *both* malware samples[9] and benign apps) so that malware detection techniques may fail to identify the mutated malware sample based on the features. Because the confusing feature values exist in both

[7] Here we take a very conservative way, because we are not sure whether this type of feature mutation works in practice.

[8] Empirically we iteratively increase x till 10; as indicated in our experiment, the program under mutation often breaks after more than 10 features are mutated.

[9] Malware samples here refer to other malware samples (likely to be evasive malware samples) instead of the one to be mutated.

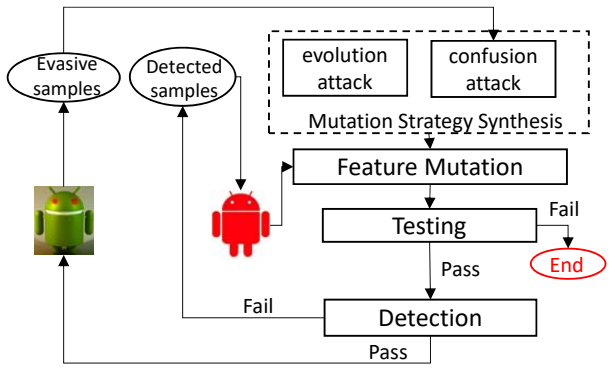

Figure 3: Illustration of mutant construction in evolution MRV. Key steps: (1) mutation-strategy synthesis; (2) program mutation/feature mutation; (3) program testing.

a malware sample and a benign app, the malware detector cannot differentiate the two apps. Therefore, the detector may mark both apps in the same category, producing false positives if the label is malware while producing false negatives if it is benign.

To find a malware feature that can actually cause confusion, we first extract RTLD features from all benign apps and malware samples in our dataset (*i.e.,* project all apps to the RTLD vector space). Then, we identify a set of (sub-) feature vectors[10] that can be projected from both benign apps and malware samples as the *confusion vector set*. For each feature vector in the confusion vector set, we count the number of benign apps that can be projected to the vector as the *confusion weight* of the vector. The rationale is that if more benign apps are projected to the vector, it is harder for the malware detector to mark the apps with this vector as malicious.

For each malware sample that we try to mutate, we first check whether its resource features appear in any vectors in the confusion vector set. If a resource feature R appears in a vector V in the confusion vector set, we then try to mutate the original feature vector of R to be the same as the vector V by mutating the contextual features. A resource feature could appear in many vectors in the confusion vector set. In our approach, we try to mutate only top 10 matching vectors ranked by the confusion weight.

If a resource feature R does not appear in any vectors in the confusion vector set, we leverage a similarity metric to find another resource feature (in the confusion vector set) R' that is most likely to be executed in the same context as R. Then we select top 10 vectors (ranked based on confusion weights) matching R' as the target vectors for mutation.

Note that there could be multiple mutated malware samples produced from mutating a single malware sample. If any of the mutated malware samples passes the validation test (Section 5) and evades the malware detection, we claim that malware confusion attack successfully produces a malware variant.

Similarity-metric computation. The aforementioned attack requires a similarity metric to find a resource feature that is most likely to be executed in the same context as another resource feature. We compute the similarity metric by the likelihood that two security-sensitive methods (*i.e.,* resource features) reside in the same program basic block. To construct the similarity metric, for each resource feature (*i.e.,* security-sensitive method m) appearing in the confusion vector set, we count its number of occurrences O_m in all apps. For each other security-sensitive method n that appears in at least one same basic block as m, we also count the number of the co-occurrences of m and n (in the same basic block) O_{mn}. Then the likelyhood that method n is invoked under the same context as method m is computed as a similarity score: $S_{mn} = \frac{O_{mn}}{O_m}$; S_{mn} also indicates the likelihood of m's context to be compatible with n. Thus, for any security-sensitive method n that does not appear in the confusion vector set, we can check for all security-sensitive methods appearing in the confusion vector set, and among these methods, pick m_i that has the highest similarity score $S_{m_i n}$ as the method that is most likely to be executed in the same context as n.

[10]If a benign app and a malware sample share over 50% of their feature values, we select their common subset of feature vectors into the confusion vector set.

4 PROGRAM MUTATION

In this section, we present how MRV mutates existing malware based on synthesized mutation strategies. The mutation process is essentially a program transformation that keeps malicious behaviors (*i.e.,* resource features) while mutating context features. To mutate context features, we develop a program transplantation framework that satisfies two needs: (a) transplanting a malicious behavior to different contexts in the same app; (b) transplanting contextual features from other apps into the existing contexts.

4.1 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) [9]. We broaden the concept of transplantation to components and methods. 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 of the organ to the host's environment.

In our transplantation framework, we take different strategies based on the type of features that need to be mutated. On one hand, to mutate temporal features or locale features of a malicious behavior (*i.e.,* resource feature), we identify (or construct) a suitable context (that satisfies the targeted value of temporal features or locale features) in the same app, and then move the malicious behavior to the identified or constructed location. On the other hand, to alter dependency features that usually require sophisticated ways (*i.e.,* specific method sequences) to achieve the desired control, we find and migrate an 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 problem of software transplantation. In the first strategy, the transplantation is intra-app. We simply save and pass the unresolved dependency and contextual information (*e.g.,* values of parameters) in the app via setting the variables and fields global. In the second strategy, although the transplantation is inter-app, we just need to transplant a program slice that contains a few dependencies. Such transplantation is lightweight compared to transplanting the whole implementation of a functional feature in previous work [9]. Intra-app transplantation is feasible for temporal and locale features because synthesizing a new entrypoint or a new component within an existing Android app results in little or no impact to other areas of the app. Mutation of dependency features requires inter-app transplantation because synthesizing new dependencies in the app is challenging. The tight coupling of dependencies brings huge

impact to other program behaviors and likely causes the mutated app to crash.

Note that although temporal features and locale features all require the transplantation of 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 the triggering part and the execution part. These two parts may not be in the same component. For example, in Figure 8, the malicious behavior of sending text message can be separated as the triggering part in the `OnCreate` method of the activity component and execution part in the `OnStartCommand` method of the 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-app transplantation, which are illustrated next.

Listing 1: Code snippet of mutated DougaLeaker malware

```
1  public void onClick(View v) {
2  User u = (User) getApplication();
3  u.androidid = Settings.Secure.getString(getContentResolver(), "
       android_id");
4  u.tel = getSystemService("phone").getLine1Number();
5  if(!isRegisterd(u.androidid)){
6  String text = "android_id = " + u.androidid + "; tel =" + u.tel;
7  Date date = new Date();
8  if(date.getHours>23 || date.getHours< 5 ){
9  android.telephony.SmsManager.getDefault().sendTextMessage(
       MyService.number, null, text, null, null); } }}
```

Listing 1 shows the mutated code related to the SMS-sending behavior in Figure 1. The mutation strategy consists of two mutations: (i) to mutate the temporal feature from lifecycle event "entering the app" (*i.e.,* `onCreate` of MainActivity) to UI event "clicking the button" (*i.e.,* `onClick` of a button's event listener), (ii) to mutate the locale feature from Service to Activity.

4.2 Inter-method transplantation

Inter-method translation refers to the migration of malicious behaviors (*i.e.,* resource features) from one method to another in the same component. Such transplantation is commonly performed to mutate the temporal features. For example, the mutation of the temporal feature in Listing 1 is inter-method transplantation (Lines 2-5 of the `onClick` method in Listing 1 are transplanted from Lines 5-8 of the `onCreate` method in Figure 1b). In the case of temporal features, the organ that needs to be transplanted is the entry of the malicious behavior and its dependencies.

First, we locate the entry of the malicious behavior. The entry of the malicious behavior is the first node on the call-graph path leading to the malicious behavior. For example, `startService` in Figure 1b is the entry of the SMS-sending behavior. In order to locate the entry of the malicious behavior, we construct the call graph from the `entrypoint` of the app (corresponding to the feature to be mutated) to the malicious method call. The entry of the malicious behavior is the first node on a call path from the entrypoint to the malicious method call (a malicious behavior could have multiple entries).

Then, we extract all dependencies related to the entry. 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 app. For example, in Figure 8, nodes 3-7 and 8 are all dependencies related to the entry (*i.e.,* node 17, `startService`). The corresponding statements are the code snippet to be transplanted.

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. For example, in Listing 1, we create an event listener and an `onClick` method to provide the temporal feature that the mutation needs.

Finally, we need to remove the organ from donor methods. If some of statements are dependent on the organ, the removal can cause the donor method to crash. To avoid the side-effects of the removal, we initialize a set of global variables with the local variables in the organ. We then replace the original dependencies on the organ by making the statements dependent on the new set of global variables. Note that in some instances, the host method is invoked after the donor method, so the set of global variables may not be initialized when the donor method is invoked. So when replacing the dependencies, we add conditional statements to check for null to avoid `NullPointerException` in the donor method. For example, after transplanting Lines 5-8 in Figure 1b, we need to remove Line 7 while keeping other lines because Lines 9-13 are control-dependent on Lines 5-6.

4.3 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. For example, the mutation of the locale feature in Listing 1 is inter-component transplantation (Lines 6-9 in the Activity component in Listing 1 are transplanted from Lines 4-7 in the Service component in Figure 1c).

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, when locale features are mutated, 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 8, 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 while maintaining the temporal feature, the regenerator needs to create ICCs to invoke the host method. To avoid crash caused by unmatching intent messages, the regenerator also adds conditional statements to avoid executing the existing code in the host method when such ICCs occur.

4.4 Inter-app transplantation

The inter-app transplantation is used to migrate the dependency feature of a malicious behavior in the donor app to the host app with an identical malicious behavior. The extraction of the dependency feature is different from migration of the triggering/execution part of the 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 paths 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 the controlling statement. To extract these necessary dependencies, we slice backward from the controlling statement until we reach entrypoints of the app. We then migrate both parts of the organ into the host app.

5 PRACTICABILITY OF ATTACKS

We take two measurements in examining the practicability of the generated attacks and filter out the impractical mutations. (a) We perform impact analysis and targeted testing to check whether the malicious behaviors have been preserved; (b) We perform robustness testing to check whether the robustness of the app has been compromised.

Impact Analysis. Our impact analysis is based on the insight that the component-based nature of Android constrains the impact of mutations within certain components. We analyze the impact propagation among components by computing the inter-component communication graph (ICCG). We find three types of components that can 'constrain' the impact of the mutations within the components themselves. If any mutations are performed in components other than these three types of components, we discard such mutations.

(i) *Isolated Component.* We define an isolated component as a component with no predecessor node or successor node in the ICCG. We can constrain the impact of mutations in an isolated component within the component for two main reasons. On one hand, because an isolated component has no successor node in the ICCG, it is guaranteed that changes in the isolated component will not break any dependencies in other components of the app (assuming no ICC through global states or variables). On the other hand, because an isolated component has no predecessor node in the ICCG, no executions of other components will lead to the isolated component. The lack of incoming edges in the ICCG suggests that the component can be invoked by only the components of other apps or system events through broadcasting intent messages that match intent-filters of the component.

(ii) *Receiving-Only Component.* We define a receiving-only component as a component with no successor node but with at least one predecessor node in the ICCG. Similar to an isolated component, changes in a receiving-only component will not break dependencies of other components. However, mutating a receiving-only component may result in crashes when other components attempt to invoke the mutated receiving-only component. MRV further performs targeted testing to ensure that the malicious behaviors are still preserved within the receiving-only component.

(iii) *Sending-Only Component.* We define a sending-only component as a component with no predecessor but with at least one successor in the ICCG. Note that we exclude the main activity component from this definition because the main activity is essential to a mobile app. A sending-only component can be mutated with manageable impacts for two main reasons. First, there is no danger of crash due to the invocation of a mutated component because sending-only components do not have incoming ICCs. Second, most components serving for the main functionality of the app should not be affected because these components can still be invoked through the main activity of the app. It is worth noting that a mutation in a sending-only component could affect the invocation of another component that has only one single incoming edge (and that edge origins from the sending-only component) in the ICCG. There are some rare cases of apps without the main activity but with services as entrypoints of the apps, and MRV would compute those services as sending-only components. For those cases, we cannot mutate the sending-only components because these components are the entrypoints of the apps.

Targeted Testing. We then perform targeted testing to further ensure that the executions are expected within these three types of components. We develop two techniques to assist the targeted testing. First, to simulate the environment where the malicious behaviors are 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 the instrumentation technique to insert logging functions at the locations of malicious method invocations. The logging functions print out detailed information about the variables, functions, and events invoked after the triggering events. We therefore attain the log files before and after the mutation under the same context (*e.g.,* the same UI or system events and same inputs). Then, we automatically compare the two log files to check the consistency of malicious behaviors.

Checking robustness of mutated apps. We leverage random testing to check the robustness of a mutated app. In particular, we use Monkey [28], a random user-event-stream generator for Android, to generate UI test sequences for mutated apps. Each mutated app is tested against 5,000 events randomly generated by Monkey to ensure that the app does not crash[11].

6 EXPERIMENT

Malware Detection 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 [44], Contagio [3], VirusShare [4], and Drebin [7]. We use VirusTotal [5] 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 3,240 apps in total. We implement the

[11] Due to the limited coverage of random testing, a mutated app that passes the testing step can still be invalid. As future work, we plan to incorporate intelligent testing techniques [42, 20] for MRV.

process of extracting RTLD features using third-party static analysis frameworks, including Soot [40] and FlowDroid [36]. 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 1,917 malware and 1,935 benign apps to perform malware detection. Our final malware dataset consists of 529 malware samples from Genome, 25 samples from Contagio, 287 samples from VirusShare, and 1,076 samples from Drebin dataset. Our final benign app dataset 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 [43, 7] 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 for each app.

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 mutate 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 (Figure 3). We choose such sequence because the transplantation goes from inter-method to inter-app as the level increases in this sequence, likely leading to a higher success rate in the program transplantation. We leave the exploration on other possible mutation sequences to our future work.

We also implement a syntactic app obfuscation tool called `OCTOPUS` similar to DroidChameleon [33]. 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 testing on the sample file (Section 5) after each obfuscation. If the testing passes, we apply the next obfuscation to the obfuscated sample (resulted from applying the current obfuscation). If the testing fails, we apply the next obfuscation to the the sample before the current obfuscation (*i.e.,* skipping the current obfuscation). In our experiment, 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 [43] 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 experiment, AppContext generates around 400,000 behavior rows on our dataset (3,852 apps), where each row is a 679-dimensional

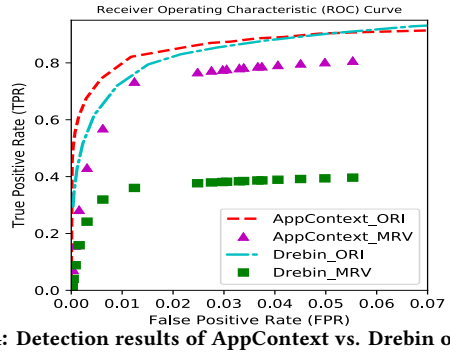

Figure 4: Detection results of AppContext vs. Drebin on the original dataset (ORI) and dataset with adversarial samples (MRV) produced by MRV

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 classifier. **Drebin** uses eight features that reside either in the manifest file or in the disassembled code to capture the malware behaviors. Since Drebin is not open source, we develop our own version of Drebin according to its description [7]. Although Drebin extracts only eight features from an app, 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. Figure 4 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 [5]. Specifically, we follow the evaluation conducted for Apposcopy [17] 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.

Learning algorithms. In our experiment, we leverage k-Nearest Neighbors (kNN), Decision Tree (DT), and Support Vector Machine (SVM) for malware detection in AppContext and Drebin. For confusion attack, we leverage Random Forest (RF) as the algorithm to train the substitute model[12]. The reason for us to use RF is that we want to use a different algorithm from the ones used in malware detection to validate our assumption in transferability [29].

Malware variants generation. We focus on generating malware variants by detected/known malware samples. Among all 1,917 malware samples, 1,739 samples can be detected by all three detection tools that we used. Because many malicious servers of malware are blocked, causing malware to crash even *before* the mutations, we test the 1,739 malware with 5,000 events randomly generated by Monkey and discard the crashed apps. This step gives us a final set of **409** valid malware samples to generate malware

[12]We optimize the parameter for SVM and DT (we use C4.5 DT [32]) using `CVParameterSelection` of Weka [6]. For RF and kNN, we tune the parameters by testing on a sample set (100 malware and 100 benign apps). We set the benign/malware ratio in each subset (of an individual tree) for RF as 3 and K value for kNN as 7.

Table 1: Number of transformable malware samples and generated malware variants by different evasive techniques and the detection results

	O.	R.	E.	C.	F.
Transformable malware	409	121	314	58	341
Generated variants	1008	212	638	58	696
Variants undetected by VirusTotal	125	113	512	53	565
Variants Undetected by AppContext	0	2	97	56	153
Variants Undetected by Drebin	0	111	460	58	518

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

variants. We then systematically apply OCTOPUS, evolution/confusion attacks, and Random MRV to all 409 valid malware samples.

7 RESULTS

7.1 Defeating existing malware detection

Table 1 shows the malware variants generated through transformation of OCTOPUS, Random MRV, malware evolution attack and confusion attack, and the detection 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 attack and evolution attack. The row "Transformable malware" refers to the number of malware samples 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 "Generated variants" shows the number of generated variants that pass the impact analysis and testing[13], and the last three rows[14] show the number of variants that can evade the detection of VirusTotal, AppContext, and Drebin, respectively.

As shown in Table 1, although the full MRV generates fewer malware variants than OCTOPUS (696 vs. 1,008), the full MRV produces much more evasive variants than both OCTOPUS and Random MRV for all three tools, especially the learning-based tools. This result indicates that the full MRV is much more effective in producing evasive malware variants than syntactic obfuscation and random transformation.

We investigate the malware variants produced by the full MRV that can still be detected by anti-virus software. We find that most variants of this kind 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.

Although originally Drebin detects more malware samples than AppContext (Figure 4), Drebin performs worse on the full MRV dataset. Given different training malware samples, the full MRV can consistently make over 60% testing variants 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.

We also notice that Random MRV becomes much more effective in evading Drebin than evading AppContext (AppContext detects almost all variants produced by Random MRV). The reason lies in the large number of syntactic features used in Drebin. 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.

One noteworthy result is that confusion attack can successfully mutate only 58 malware samples into working malware variants. The reason is that confusion attack usually requires mutating more contextual features than evolution features. We observe in our experimental data that the likelihood of an attack to break the app increases as the number of mutations in the attack increases. Actually, confusion attack synthesizes more than 1,000 variants, and most of the variants are unable to run. However, such conversion rate is already high compared to Random MRV. Random MRV generates more than 320,000 variants, but only 212 of them can run without crashing (and only 2 can evade the detection of AppContext). Such result suggests that considering the feasibility of an attack is essential in generating adversarial malware samples.

[13]The variants are generated at each level, and one malware sample may result in multiple malware variants.

[14]For AppContext and Drebin, we show the number of variants that cannot be detected by models based on all training algorithms.

Table 2: 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

7.2 Effectiveness of attacks at each level

For evolution attack, we also investigate the effectiveness of mutation at each RTLD level. Table 2 shows the detailed detection results of evolution attack at each mutation level.

Table 2 shows some interesting observations. For example, for anti-virus software and Drebin, the level that produces the largest number of evasive variants is on the locale-feature level, while for AppContext, the level that produces the largest number of evasive variants is on the dependency-feature level. This result indicates that mutating at the locale-feature level is more effective for 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 experiment (*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.

We also observe that most of unsuccessful variants produced at the dependency-feature level are due to the fact that a malicious behavior cannot be triggered in the simulated testing environment. The reason of lacking triggering is that by transplanting conditional statements from one component/method to another component/method, the internal logic of the original malware sample is broken. Some of the transplanted conditional statements may be mutually exclusive with the existing conditions in the code, thus making the malicious behavior infeasible to be triggered. As an ongoing effort, we plan to leverage a constraint solver to identify the potential UNSAT conditions when synthesizing mutation strategies.

7.3 Strengthening the robustness of detection

We also investigate the possibility of leveraging variants produced by MRV to strengthen the robustness of detection. We propose the following three techniques.

Adversarial Training. We randomly choose half of our generated malware variants into the training set to train the model, and put the other half of generated variants into the testing set to evaluate the model[15].

Variant Detector. We create a new classifier called variant detector to detect whether an app is a variant derived from existing malware. The variant detector leverages *mutation features* that are generated from each pair of apps' RTLD features to reflect the feature differences between the two apps. The number of mutation features is the same as the number of RTLD features. The difference is that for any RTLD feature that the two apps disagree on, the mutation feature (corresponding to the RTLD feature) is the (bidirectional) mutation between the apps on the RTLD feature. If the pair of apps are derived from same malware, we label the feature vector as "variant". Otherwise, we label the feature vector as "unrelated". Because only a small portion of all pairs of apps would have a "variant" relation, the trained model would be biased to the majority class (*i.e.,* the "unrelated" class). To resolve such issue, we use SMOTE [14] to make both classes to have an equal number of instances by creating synthetic instances. We then use the trained model on each app labeled (by malware detectors) as benign. For each of the apps, we create pairs to produce mutation features by grouping the app with each malware sample in our training set. Then the trained model determines whether the app is a variant of malware in the training set based on the mutation features.

Weight Bounding. We constrain the weight on a few dominant features to make feature weights more evenly distributed. For example, in the case of SVM, we constrain w in the cost function of SVM:

$$\min_{w \in \mathcal{R}^d} \| w \|^2 + C \sum_{i}^{N} \max(0, 1 - y_i f(x_i))$$

We observe that adversaries can produce evasive malware variants by applying just a few mutations on dominant features in contrast to many more mutations on other non-dominant features. Therefore, to locate dominant features, we select all 44 malware variants produced by fewer than three mutations, and summarize 17 dominant features that enable the production of the variants. To compute the specific range of the weight, we put only 44 malware variants and their original malware samples as malicious samples in the training set, and record the range value of the weight of the 17 dominant features under different parameters. We then summarize the constraints in reasonable settings (TPR $\geq$ 0.80 and FPR $\leq$ 0.10) and put the hard constraints in the training phase.

Figure 5 presents the detection results of AppContext's malware detection[16]. The red line represents the detection performance on the original dataset, and the purple triangles represent the detection performance on the dataset with malware variants produced by MRV. The other three curves represent the detection performance of three proposed protection techniques on the dataset with malware variants. As shown in Figure 5, all three proposed techniques can alleviate the MRV attacks. The variant detector technique can reach almost the same performance as the original malware detector (while being more secure/robust to malware variants).

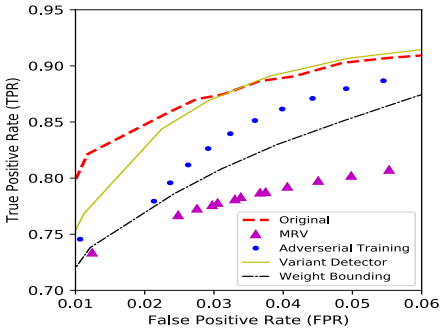

Figure 5: Detection results of AppContext (SVM) when different defense mechanisms are applied

Table 3: Number of malware samples evading detection of AppContext or Drebin under different algorithms

Detector	ORI.	AT.	VD.	WB.
AppContext	178	125	106	152
Drebin	38	19	8	23

ORI. = Original detection, AT. = Adversarial training
VD. = Variant detection, WB. = Weight bounding

To alleviate the concerns that our proposed defenses are overfitting to MRV attacks, we also investigate whether the trained models can assist detecting not only malware variants but also unknown malware samples in general. We choose to investigate the malware samples evading the detection of the original AppContext and Drebin (178 and 38 malware samples evade the detection, respectively)[17]. As shown in Table 3, all the protection mechanisms can help detect evasive malware samples, and only eight of the samples can evade the detection of the variant detector technique.

8 CONCLUSION

In this paper, we have proposed practical attacks that mutate malware variants to evade detection. The core idea is to leverage existing malware program structures to change the features that are non-essential to malware but important to malware detectors. To achieve this goal, we have presented the MRV approach including static analysis, phylogenetic analysis, machine learning, and program transplantation to systematically produce new malware mutations. To the best of our knowledge, our work is the first effort toward solving the malware-evasion problem by altering malware bytecode without any knowledge of the underlying detection models.

MRV opens up intriguing, valuable venues of applications. First, the proposed attacks can be used to evaluate the robustness of malware detectors and quantify the differentiability of features. Second, MRV can help discover potential attack surfaces to assist the iterative design of malware detectors. Finally, the program transplantation framework (capable of changing malware features)

[15]We perform ten-fold cross-validation in our experiment to report TP and FP.

[16]We present only SVM-based model here due to the limited space. Other learning algorithms present similar patterns as SVM. We leave investigation of specific differences across models as future work.

[17]All the numbers are counted when the false positive is within 0.06.

can be written as a malicious payload within malware and such adaptive malware are valuable for the community to investigate.

ACKNOWLEDGMENT

This work is supported in part by NSF grants CNS-1223967, CNS-1513939, CNS-1330491, CNS-1434582, CNS-1564274, and CCF-1409423.

REFERENCES

[1] Airpush Detector. https://goo.gl/QVn82.
[2] Airpush Opt-out. http://www.airpush.com/optout/.
[3] Contagio. http://contagiominidump.blogspot.com/.
[4] Virusshare. http://virusshare.com/.
[5] Virustotal. https://www.virustotal.com/.
[6] Weka 3: Data mining software in Java. http://www.cs.waikato.ac.nz/ml/weka/.
[7] D. Arp, M. SPreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: effective and explainable detection of Android malware in your pocket. In *Proc. NDSS*, 2014.
[8] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *Proc. CCS*, pages 217–228, 2012.
[9] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proc. ISSTA*, 2015.
[10] A. D. Baxevanis and B. F. F. Ouellette. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. John WileySons, 2004.
[11] 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 *Proc. KDD*, pages 387–402, 2013.
[12] B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE Trans. TKDE*, pages 984–996, 2014.
[13] A. A. Cárdenas and J. S. Baras. Evaluation of classifiers: Practical considerations for security applications. In *Proc. AAAI Workshop Evaluation Methods for Machine Learning*, pages 777–780, 2006.
[14] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of artificial intelligence research*, pages 321–357, 2002.
[15] 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 *Proc. USENIX Security*, pages 659–674, 2015.
[16] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on Android malware detection. *IEEE Trans. TDSC*, 2017.
[17] Y. Feng, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proc. FSE*, pages 576–587, 2014.
[18] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proc. NIPS*, pages 2672–2680, 2014.
[19] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.
[20] S. Hao, B. Liu, S. Nath, and R. Govindan. PUMA: Programmable UI-automation for large-scale analysis of mobile apps. In *Proc. Mobisys*, pages 204–217, 2014.
[21] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983*, 2017.
[22] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. Adversarial machine learning. In *Proc. AISec*, 2011.
[23] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *KDD*, pages 1357–1365, 2013.
[24] H. W. Kuhn and B. Yaw. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, pages 83–97, 1955.
[25] P. Legendre and L. Legendre. *Numerical Ecology: Developments in Environmental Modelling*. Elsevier., 1998.
[26] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. In *Proc. ICLR*, 2017.
[27] Montrojans, ghosts, and more mean bumps ahead for mobile and connected thingskey. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2017.pdf.
[28] Monkey. http://developer.Android.com/tools/help/monkey.html.
[29] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.
[30] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against deep learning systems using adversarial examples. In *Proc. ASIACCS*, 2017.
[31] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *Proc. Euro S&P*, pages 372–387, 2016.
[32] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1986.
[33] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android anti-malware against transformation attacks. In *Proc. ASIACCS*, pages 329–334, 2013.
[34] N. Rndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. IEEE S & P*, pages 197–211, 2014.
[35] 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 *Proc. ACSAC*, pages 81–90, 2015.
[36] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proc. PLDI*, pages 43–54, 2015.
[37] A. Singh, A. Walenstein, and A. Lakhotia. Tracking concept drift in malware families. In *Proc. AISec*, pages 81–92, 2012.
[38] C. Smutz and A. Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *Proc. NDSS*, 2016.
[39] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
[40] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *Proc. CASON*, 1999.
[41] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proc. NDSS*, 2016.
[42] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Proc. FASE*, pages 250–265. 2013.
[43] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. ICSE*, pages 303–313, 2015.
[44] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. IEEE S & P*, pages 95–109, 2012.

APPENDIX A MALWARE EVOLUTIONS

To further assist the improvement of existing detection, we summarize the evolution patterns of malware for contextual features (Section 3). We infer malware evolution patterns from phylogenetic trees. Figure 7 demonstrates how 32 samples are evolved in the `AdDisplay` family, and Figure 6 demonstrates how 28 samples are evolved in the `Droid_Kunfu` family, where the number labeled in the bottom of each phylogenetic tree denotes the distance between two nodes. A node in a phylogenetic tree could be a leaf node that denotes a malware sample, and also could be an internal node that denotes a cluster grouped from its children nodes. Figures 10 and 11 show detailed introduction of each malware sample in the families.

Temporal features. Among all temporal features, lifecycle events are most likely to evolve. Such evolution can be intra-component (*i.e.,* the entrypoint is mutated to another method in the same Android component) or inter-component (*i.e.,* the entrypoint is mutated to a method in a different Android component). For inter-component evolution, the corresponding ICCs have been added to incorporate the evolution.

Malware also frequently evolve temporal features of 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 triggered by third-party ad-network intents (*e.g.,* `com.airpush.android.PushServiceStart`) also evolve[18].

[18]Such evolution is due to that the emerging `Adblock` apps (*e.g.,* Airpush Detector [1], `Airpush` Opt-out [2]) force Adware authors to implement malicious behaviors by themselves instead of leveraging third-party libraries such as `Airpush`.

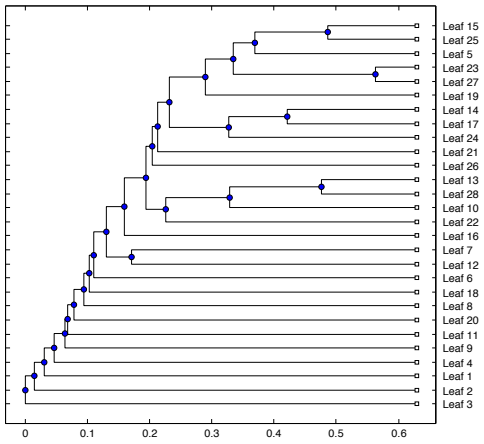

Figure 6: Phylogentic tree generated for the `DroidKungFu` family. Each leaf in the graph denotes a malware sample in `DroidKungFu` family, where leaf nodes (1–4) belong to `droidkungfu.ab`, (5–9) belong to `droidkungfu.aw`, (10) belongs to `droidkungfu.bb`, (11-12) belong to `droidkungfu.bl`, (13-15) belong to `droidkungfu.c`, (16–22) belong to `droidkungfu.g`, (23-28) belong to `droidkungfu.m`.

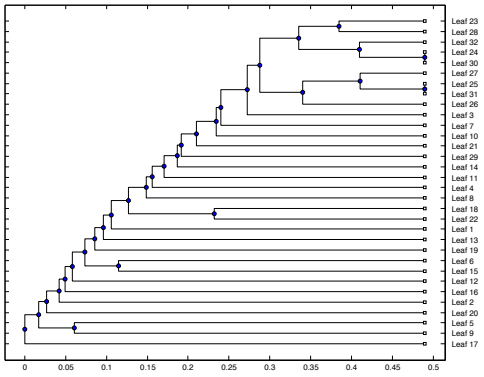

Figure 7: Phylogentic tree generated for the `addisplay` family. Each leaf in the graph denotes a malware sample in the `addisplay` family, where leaf nodes (1–9) belong to `addisplay.adswo`, (10–23) belong to `addisplay.airpush`, (24–25) belong to `addisplay.dowgin`, (26) belongs to `addisplay.kuguo`, (27–28) belong to `addisplay.waps`, (29–32) belong to `addisplay.wooboo`.

Locale features. We observe that locale features often evolve with temporal features. The reason is that when malware authors mutate an entrypoint of a malicious behavior from one component to another, instead of adding another ICC from the new component to invoke the code in the original component, the malware authors tend to directly migrate the code to the new component. Most of such evolutions occur for malicious behaviors with fewer (*e.g.,* one or two) entrypoints, and the corresponding security-sensitive methods are usually directly invoked in the entrypoint methods. We find that the most frequent change of locale features is migration from an activity or receiver component to a service component. However, we observe fewer evolutions the other way around (*i.e.,* evolutions from a service component to an activity or receiver component). Such pattern is due to that 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).

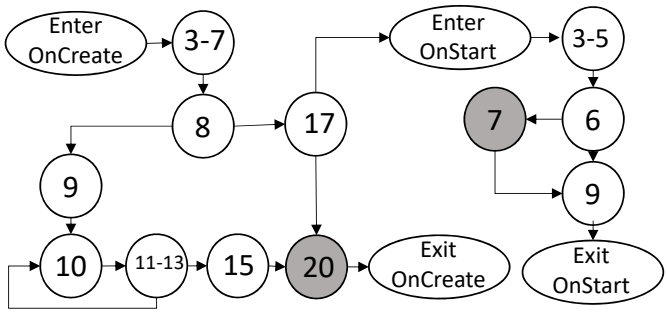

Figure 8: Inter-Procedural Control Flow Graph of DougaLeaker

Dependency features. The most common evolution for dependency features is adding controls of a security-sensitive method (*i.e.,* a resource feature) through ICCs. Such evolution can confuse malware detection techniques because the generated program dependence graphs can reflect only the control dependencies between a malicious behavior and values stored in Intent messages (of ICCs). Connecting the control dependencies between the malicious behavior and the original value (*e.g.,* the current system time) requires precise analysis of ICC. However, the current use of such analysis is absent or limited.

The other frequent evolution is making malicious behaviors dependent on attributes of external entities (including `Internet connection` and `telephony manager`). The Internet connection reflects the command & control behaviors through network servers. The package manager and telephony manager suggest that malware control malicious behaviors 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 a number of interesting cases that combine the mutations 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 sample leverages the system event `android.net.conn.CONNECTIVITY_CHANGE` to get notified when the network connection changes so the malware sample can obtain the `NetworkInfo` from the system Intent message for new network connections. To evade detection, the malware sample evolves to leverage the system event `android.intent.action.USER_PRESENT`. Such event gets the malware sample notified when a user is present after unlocking the screen. Then the malware sample starts a service and uses a timer in the service to repetitively invoke `getNetworkInfo` every 20 minutes. In this way, the malware author mutates the original value of the temporal feature (`CONNECTIVITY_CHANGE`) to a value (`USER_PRESENT`) completely unrelated to network connection. Meanwhile, the locale feature value is mutated from `receiver` to `service`, and the control dependency with a timer is also added. Malware variants produced by MRV can be especially helpful in detecting such sophisticated evolution.

APPENDIX B IDEAS BEHIND OUR ATTACKS

An illustration. The goal of malware detection is to classify an app as "malware" or "benign". In MRV, we achieve the goal by mutating the feature vector 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 4). In order to evade detection, we can mutate feature values V_i in three strategies:

(i) *look-alike-benign-app*: mutating V_i to be exactly the same as the feature vector of a benign app, *e.g.,* $\langle a, b, c'\rangle$ of B_1 in Table 4;

(ii) *look-alike-misclassified-malware*: mutating V_i to be exactly the same as the feature vector of a malware being misclassified as benign, *e.g.,* $\langle a, b, c'\rangle$ of M_2;

(iii) *look-alike-unclassifiable-app*: mutating V_i to the feature vector 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.

Malware Evolution Attack: 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 4). Such attack follows the idea of the aforementioned strategy of look-alike-unclassifiable-app (iii), but as the attack is derived from existing mutations, the likelihood of the mutations to break the malicious behaviors decreases (as confirmed by our empirical evaluation). Note that we do not come up with any attack that conforms to the aforementioned strategy of look-alike-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 has a high likelihood to break malicious behaviors and eliminate the maliciousness.

Malware Confusion Attack: In Table 4, malware sample M_2 shares the same feature vector with benign app B_1, and the malware detector cannot tell the difference and therefore marks both apps in the same category (producing false positives if the label is malware while producing false negatives if it is benign). This attack follows the idea of the strategy look-alike-misclassified-malware (ii) but does not rely on the detection result from any particular detection algorithm.

Table 4: 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'	Benign

Columns f_1 to f_3 are three feature columns.

Attacking Weak Spots 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 any given malware-detection technique as V (shown in the Venn diagram in Figure 9).

The differentiability of selected features can be represented by the intersection of the vector space (denoted as B) for the existing benign apps and that (denoted as M) of the 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 sample 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 positives 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.

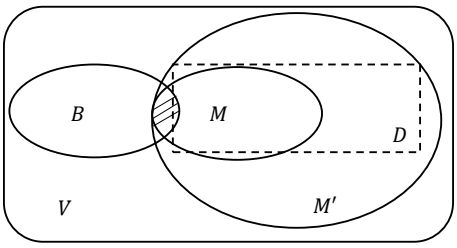

Figure 9: A feature vector space V, the feature vectors of existing benign apps B, the feature vectors of existing malware M, the feature vectors that can be detected by detection model D, the feature vectors of all potential malware M', and their relationships.

Feature evolution attack is 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 of the variants fall into the area of $M' \setminus D$). Feature evolution attack mutates RTLD feature values iteratively at each level (following the sequence of temporal feature, locale feature, and dependency feature).

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 model[19]. 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 to detecting only 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 new malware variants produced through known mutations. Such mutations may employ not only syntactic and semantic obfuscation techniques, but also feature mutations based on analyzing the evolutions of malware families.

Feature confusion attack is based on the insight that malware detection usually performs poorly in differentiating malware and benign apps with the *same* feature vector. As discussed earlier, if we simply mutate malware feature vectors to benign feature vectors (*i.e.,* feature vectors in space B), such mutation would

[19]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 the same as some malware samples.

Figure 10: The detailed (ind, apk name, and its corresponding subfamily) items in the `AdDisplay` family phylogenetic analysis.

Figure 11: The detailed (ind, apk name, and its corresponding subfamily) items in the `DroidKungFu` family phylogenetic analysis.

generally break or weaken the malicious behaviors (*i.e.,* turning malware into benign apps). So, our design decision is making malware with unique malicious feature vectors (*i.e.,* $M \setminus (B \cap M)$) to possess the feature vectors same as benign apps (*i.e.,* $B \cap M$). Because some apps 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 behaviors in the host malware.

Threat Model & Use Cases. We assume that an attacker has only black-box access to the malware detector under consideration. Under such assumption, the attacker can feed any malware sample as the input to the detector 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 detector. The attacker is capable of manipulating the malware's binary code, but has no access to the malware's source code. We assume that the attacker has access to the existing malware samples (*i.e.,* samples that are correctly detected by the malware detector), and the goal of the attacker is to create malware variants with the same malicious behaviors, but can evade the detection.

Although we present our techniques 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 a detector model is (*i.e.,* the larger D is), the more difficult for a mutated malware to evade the detection (*i.e.,* the smaller $M' \setminus D$ can be). *Feature confusion attack* can be used to evaluate the differentiability of selected features. The more differentiable a feature is, the less the opportunity is for a malware sample to confuse the detector (*i.e.,* smaller $B \cap M$ is desirable).

Discussion. *Limitations of app testing.* In testing processes for transformed apps, the testing checks only two main things: whether the app 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. Such issue could be addressed by manually creating regression tests for the malicious behaviors. Additionally, it is infeasible to test whether malware become less profitable or easier to be detected by manual inspection.

Adware threat in the experiment. In our experiment, we view one family of adware (*i.e.,* AdsDisplay) as a malware family due to the fact that both malware and adware are "unwanted software", and they both differ significantly from benign apps according to a recent study [35].