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

***Abstract*—Existing work on detecting Android malware employs either signature-based or learning-based techniques to extract features that represent common malicious behaviors. However, most (if not all) of these defenses suffers from two inherent limitations. First, there is a lack of differentiability: selected features may not reflect essential differences between malware and benign apps. Second, there is a lack of comprehensiveness: the detection signatures/models are usually based on prior knowledge of existing malware (i.e., training dataset) so malware can evolve to evade the detection.**

To address these limitations, 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 distinct types of variation strategies to target malware evolution attacks and confusion attacks, respectively. 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 the malware bytecode to generate new malware variants.

Our approach produces malware variations that can have high likelihood to evade detection while still retaining their malicious behaviors. Our empirical evaluation on 3,852 Android benign apps and malware (originated from more than 35 families) shows that MRV can effectively produce malware variants that can evade detection techniques based on specified signatures or learned patterns (ranging from 23.8% to 74.4% evasion rate for various detection techniques). To counter this type of attack, we provide insights on how to improve the differentiability and comprehensiveness of current defense techniques.

I. Introduction

Along with the exponential growth on markets of mobile applications (*apps* in short), comes the frequent occurrence of mobile malware on app markets. According to the McAfee Security Report 2016[1], there are more than 37 million mobile malware samples 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 learns discriminant features from analyzing information flows and semantics of malware.

One major challenge for both signature-based and learning-based malware detection approach is to form an informative feature set for signature or detection model. To address challenge, existing malware detection tends to include as many features as possible. For example, Drebin, a recently published malware detection work [6], uses the feature set containing 545,334 features. Recent study [51] shows that such large feature set has numerous non-informative or even misleading features. Therefore, in this paper, we investigate the question: *can a malware be mutated to evade detection by changing its feature values while maintaining its malicious behaviors*[2]*?* More formally, we name such "mutations of malware based on feature values" as *Malware Recomposition Variation (MRV)*.

A key observation made in our research is that, features, which abstract concrete malicious behaviors, are fragile, and they could be easily mutated (*i.e.,*, changed). The susceptibility of such features makes it possible to evade detection if malware are properly mutated [17], [50], [58]. Our research suggests that *features that are unique to malware are not necessary needed for forming malicious behaviors*. Such result is mainly due to two factors.

First, learning-based detectors often confuse non-essential features (*i.e.,* features that are not essential for forming malicious behaviors) in code clones as discriminative features. Copy-paste practice is prevalent in malware industry which result in many code clones in malware samples [21]. 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 appeared in many malware samples but not in benign apps). Learning-based detector place higher weight on these features not because these features are essentail to malicious behaviors but because these features appeared in malware much more frequently than in benign apps. Adversaries could simply leverage such fact to mutate some of these non-essential features with higher weight in detecting model to evade detection.

Second, the features essential to malicious behaviors are different for each malware family. Almost all existing learning-based malware detection using a universal feature set to detect malicious samples for all malware families. However, based on recent research result [67] mined from 1,068 research papers and malware documents, each malware family associates with a distinct set of malware behaviors and concrete features. Using a universal set of features for all malware families would result in a large number of non-essential features to characterize each family. Furthermore, as previously mentioned, if these non-essential features are unique in some malware samples, the trained detection model can be evaded by mutation the value of the non-essential features.

In this paper we focus on synthesizing mutation strategies (*i.e.,* what kind of features we should mutate to evade detection) and automating program transformation (*i.e.,* how to apply mutations on malware bytecode to ensure the robustness of the app while preserving malicious behaviors). Different from existing work [17], [50], [58], we explore the capability of attackers in a more realistic attacking scenario: the attackers

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

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

can feed any app as the input to the detector and know the binary detection result (*i.e.,* detected as malware or not) without any additional information.

MRV should satisfy three requirements (C1–C3).

C1: Evading Malware Detectors. To evade a malware detection model, an adversary need to identify the non-essential features and compute the mutated feature value that can evade detection. This usually requires an adversary to possess internal knowledge and understanding of malware detectors. Unfortunately, generally an adversary may have little (or even no) knowledge about the malware-detection model (such as features and algorithms). Moreover, the particular knowledge to a single malware-detection model is too specific and conducting MRV is unlikely to succeed, especially if the model (*e.g.,* the one in VirusTotal [5]) is based on combining multiple techniques.

C2: Preserving Malicious Behaviors. The mutated malware should maintain the original malicious purposes and therefore simply converting malware's feature values to another app's feature values is likely to break the malicious . For example, the malicious behaviors are usually designed to be triggered under certain contexts (to avoid user attention and gain maximum profits [61]), and the controlling logic of the malware is too sophisticated (*e.g.,* via logic bombs and specific events) to be changed.

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

To ***evade malware detectors (C1)***, we propose two mutation strategies, *Malware Evolution Attack* and *Malware Confusion Attack* (Section IV), based on a feature model named as RTLD (Section III), that generally reflects the susceptibility of a detection technique to the mutations of malware feature values. This model is based on the insight that malware detection techniques not only leverage essential features of malicious behaviors (*e.g.*, security-sensitive method calls), but also incorporate other contextual factors (*e.g.*, when and where a method is invoked) to reflect the malicious intentions [25], [61], [65]. Additionally considering contextual factors is due to that the same values of such essential features exhibit in both malware and benign apps, so using only essential features is typically insufficient to differentiate malicious behaviors from benign ones. The proposed RTLD feature model summarizes both essential and contextual features in mobile apps, and categorizes them in four aspects (*i.e.,* **R**esource, **T**emporal, **L**ocale, and **D**ependency). Note that as a proof of concept, the RTLD model used in this paper is only an approximation of feature space used in malware detectors, so it is likely that the RTLD model would miss some features and thus MRV neglects to mutate those features. Our empirical results (Sec. VIII) show that MRV based on our RTLD model is already able to mutate a non-trivial portion of malware samples, demonstrating the effectiveness of the model. Actually, we expect that MRV would produce better results given a more comprehensive feature model.

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

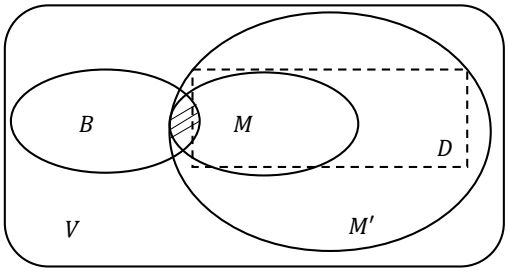

Fig. 1: A feature vector space V, the feature vectors of existing benign apps B, the feature vectors of existing malware M, the feature vectors can be detected by detection model D, the feature vectors of all potential malware M' and their relationships.

To ***maintain the robustness of apps (C3)***, we develop a new technique, inspired by program transplantation [43], [53], to reuse the existing implementations of desired features instead of randomly mutating or synthesizing the code. In particular, we develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation[3]. By leveraging the existing implementations, this technique enables systematic and automatic mutations on malware samples while aiming to produce a well-functioning app.

Main Contributions. This paper makes the following main contributions.

- ***Observation.*** We identify differentiability of selected features and robustness of detection models as two fundamental limitations of malware detection, from which we demonstrate the feasibility of producing effective attacks (Sec. II).
- ***Characterization.*** We propose an RTLD feature model that characterizes and differentiates contextual and essential features of malicious behaviors (Sec. III).
- ***Attacks.*** We propose malware recomposition variation (MRV) to produce two attack types (*feature evolution attack* and *feature confusion attack*) to effectively mutate existing malware for evading detection (Sec. IV).
- ***Framework.*** We develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation to automatically mutate app features (Sec. V).

II. MRV Design

A. Limitations 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 1).

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

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

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

B. Threat Model

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.

C. Overview of MRV

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

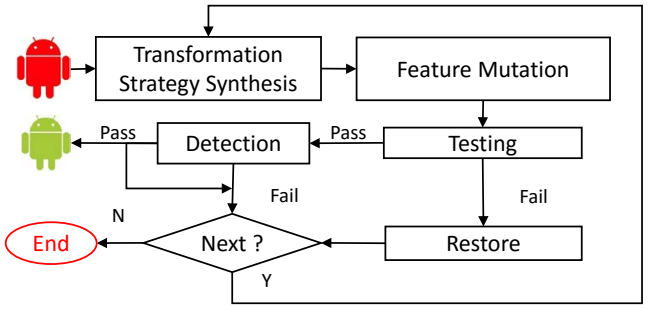

Fig. 2: Illustration of mutant construction in evolution MRV. Key steps: (1) transformation-strategy synthesis; (2) program transformation/feature mutation; (3) program testing.

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

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

Use Cases. 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 the detector model is (*i.e.,* the larger D is), the more difficult for a mutated malware to evade detection (*i.e.,* the smaller $M' \setminus D$ can be). The detail of each step is

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

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

elaborated in subsequent sections. *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 to confuse the detector (*i.e.,* smaller $B \cap M$ is desirable).

III. RTLD Feature Model

Malware features in existing detection. Existing malware detection techniques mainly use two types of features: syntactic features and semantic features. Syntactic features include opcode sequences [32], constant network addresses, filtered intents, declared hardware components, third-party packages [7], *etc*. Based on extracting only external symptoms, these features, however, do not embody the rich semantic information inherent in malware programs, and therefore could be easily circumvented using simple program obfuscations such as metamorphism and polymorphism, being thoroughly studied in previous work [49].

Compared with the aforementioned types of syntactic features, semantic features are more difficult to obfuscate, and can thus be used as robust features for understanding the malware evolution and detection of malware. Hence, in this work, we narrow down the scope of our focus to "semantic" features due to the fact that it is quite simple to evade a detection tool based on syntactic features. Semantic features such as control-flow graphs [20], weighted API-dependency graphs [64], program information flow graphs [27] have been used for malware classification and analysis.

In this work, we characterize semantic features using our proposed RTLD 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. Contextual features are used in malware detection because using only essential features cannot differentiate malware from benign apps. For example, a behavior of sending an SMS message is malicious when it is performed in the background stealthily without awareness of users. In contract, if an SMS message is sent when the user clicks a "send" button, the behavior is likely a benign behavior.

The RTLD feature model covers four main aspects:

- **Resource.** What security-sensitive resources are obtained by malicious behaviors?
- **Temporal.** When are the malicious behaviors triggered?
- **Locale.** Where do the malicious behaviors occur?
- **Dependency.** How are the malicious behaviors controlled?

In this work, as a proof of concept, we leverage the call graph and inter-procedure control flow graph to extract and identify the RTLD features. However, the RTLD feature model itself is general and can be applied to the features extracted through other means. We use the simplified code snippet of the DougaLeaker malware[6] shown in Figure 3 to illustrate the feature model. The code snippet shows two malicious behaviors of the DougaLeaker malware. First, the malware saves the Android ID and telephone number of the victim device to global class `User` when the app starts (Lines 5-7 in Figure 3b). Then, the malware reads the contacts on the victim device (Lines 8-13 in Figure 3b) and sends the contacts to a malicious server (Line 20 in Figure 3b). The malware

[6]MD5 of the malware is e65abc856458f0c8b34308b9358884512f28bea31fc6e326f6c1078058c05fb9.

```
1  public class User extends Application{
2    public String androidid;
3    public String tel;
4  }
```

(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
21   }
22 }
```

(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;
10   }
11 }
```

(c) `MyService` of DougaLeaker malware

Fig. 3: Motivating Example: DougaLeaker malware

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. In practice, the resource features are closely linked to finishing one task, and therefore, we present them together according to different malicious behavior types:

• *Access to permission-protected resources*. Permission-protected resources are essential to malicious behaviors. Malware usually use Android APIs to get the privileges or sensitive information. Therefore, we leverage static analysis techniques [11], [31], [55] to find locations of the API method invocations (*e.g.*,, `sendTexeMessage` in Figure 3c).

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

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 callgraph algorithm implemented in Soot [55]. 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 3) in `exec_post` and `sendTextMessage` method invocation (Line 7 in Figure 3c) in `onStartCommand` in the call graph. Due to space limit, we omit many details here. For the detailed algorithm that we used for extracting RTLD features, please refer to our accompanying technical report[8].

The **temporal** features describe the contexts when the malicious behaviors are triggered. In particular, we characterize the events for triggering malicious behaviors as follows.

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

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

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

To extract 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 communication). For detailed analysis, please refer to our accompanying technical report[8].

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

The locale features in general reflect the visibility 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`), then the permission use is running in the background, and also it is a continuous task (even after exiting the app). The detail of extracting locale features can be found in our accompanying technical report[8].

The **dependency** features describe the control dependencies of the invocation of the malicious behavior. 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, the `DroidDream` malware 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 executions at late night when users are likely sleeping.

Usually, in malware detection, such dependencies are represented by external events or attributes that can affect control flows to the security-sensitive behaviors. The dependency features are generated from program dependence graphs (PDGs) obtained in static analysis of apps.

In particular, we construct inter-procedure control-flow graph (ICFG) to extract dependency features. Based on the ICFG, we construct the subgraphs from each entrypoint to the 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 ex-

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

[8]https://dl.dropboxusercontent.com/u/37908210/MRV-TR.pdf

TABLE I: 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.

ecutions, 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 5 shows the ICFG of the `onCreate` and `onStartCommand` methods. As shown in the Figure, the `sendTextMessage` method in `onStartCommand` (Line 7) is controlled by the conditional statement on Line 6 in `onStartCommand` and the conditional statement on Line 8 in `onCreate`. 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.

IV. Mutation Strategy Synthesis

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

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

We propose *Malware Evolution Attack* to mimic the feature mutations in the space of actual malware evolution in the real world. This attack is based on the observation that evolution of a malware is actually the process of mutating its corresponding feature values to reach "*blind spots*" of malware detection. For example, we can follow the feature mutations from M_1 to M_2 (*i.e.,* $c \rightarrow c'$ in f_3) to derive a new malware variant M_v from M_3 (shown in Table I). In particular, we conduct *phylogenetic analysis* on each family of malware and derive mutation strategies based on the phylogenetic tree of each malware family. Such attack follows the idea of the aforementioned strategy of look-like-unclassifiable-app (iii)[9], but as the attack is derived from the existing mutations, the likelihood of the mutations to break the malicious behaviors decreases (as confirmed by our empirical evaluation). Interestingly, we find that some of the mutation strategies effectively generate evasive malware variants when applied across families (*i.e.,* using the evolution history of a malware family to guide the mutation of malware instances from a different family).

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

We present our techniques of synthesizing strategies to mutate program features that achieve two main goals:

- Goal 1: the mutation strategies should enable the mutated malware to evade the detection of existing techniques (*i.e.*, reach the "blind spot" of existing detectors).
- Goal 2: the mutation strategies should preserve the malicious behaviors of the original malware (*i.e.*, achieve the same malicious behaviors without crashing).

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

A. Evolution Attack vs. Confusion Attack

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

In *informative scenarios*, adversaries know the type of algorithms used in the detection, and therefore we develop

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

the targeted attack called **confusion attack**. The main idea of malware confusion attack is to mimic the malware that can generally evade detection, *i.e.,* confusing the malware detectors by modifying the feature values that can be shared by malware and benign apps. Such defeating mechanism is based on the observation that malware detectors (based on a classifier) could be easily misled by feature omissions and substitutions.

B. Algorithm Details

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

(i) To generate **evolution attack**, we identify a feature set called *evolution feature set*. In the set, each feature is evolved either at intra-family level or inter-family level. For each feature vector in the *evolution feature set*, we count the number of evolutions as *evolution weight*, where *intra-family evolution weight* is proportional to the number of evolutions at intra-family level, and *inter-family evolution weight* is proportional to that at inter-family level. The rationale is that if the feature type has already been evolved frequently under observation, it is more likely to be evolved according to the nature of the law (in biological evolution process [12]). Figure. 4 shows how 32 samples are evolved in the `AdDisplay` family[10].

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

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

(iii) If a resource feature R appears in a vector V in the critical feature set, we then mutate the original feature vector of R to be the same as vector V by mutating the contextual features. A resource feature could appear in many vectors in the critical feature set. Then we mutate top K matching vectors ranked by the corresponding evolution or confusion weight.

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

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

[10]The number labeled in the bottom of each phylogenetic tree denotes the distance between two nodes. The node could be a leaf node for denoting a malware, and also could be an internal node for denoting a cluster grouped from its children node. The Hungarian-type algorithm [41] is used to compute the malware distance based on the RTLD features.

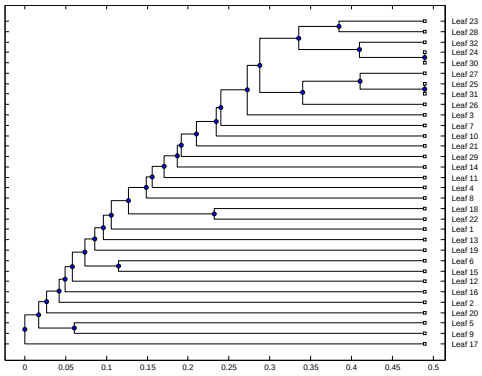

Fig. 4: Phylogentic tree generated for `addisplay` family. Each leaf in the graph denotes a malware in `addisplay` family, where leaf nodes (1–9) belong to `addisplay.adswo`, (10–23) belong to `addisplay.airpush`, (24–25) belong to `addisplay.dowgin`, (26) belongs to `addisplay.kuguo`, (27–28) belong to `addisplay.waps`, (29–32) belong to `addisplay.wooboo`.

We define *the similarity metric* as the likelihood that two security-sensitive methods (*i.e.,* resource features) reside in the same program basic block. The reason that we choose a basic block instead of a coarser-granularity unit such as a function is that context features also include control dependencies. If two method invocations are in the same function body but not the same basic block, the execution contexts could be different (*e.g.,* a basic block can be invoked only during 11PM-5AM). For each security-sensitive method m appearing in the confusion vector set, we count the number of its occurrences O_m in all apps. Similarly, for another security-sensitive method n that appears at least once in the same basic block as m, we count the co-occurrences of m and n in the same basic block as O_{mn}. Then the likelihood that method n is invoked under the same context as method m is given by $S_{mn} = \frac{O_{mn}}{O_m}$. Such equation computes the likelihood that m's context is compatible with that of n. Going back to case (iv), for any security-sensitive method n that does not appear in the *critical feature set*, we can select method m_i (a) appearing in the critical feature set, (b) having the highest similarity score $S_{m_i n}$ as the one probably to be executed in the same context as n.

V. Program Mutation

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

A. Transplantation framework

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

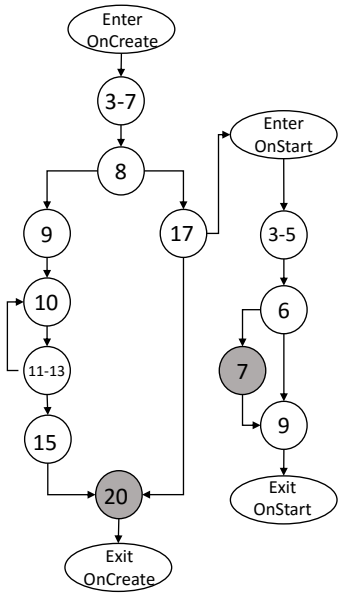

Fig. 5: Inter-Procedural Control Flow Graph of DougaLeaker

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 the temporal features or locale features of the program, due to the fact that the dependency of the malicious behavior on the program is usually simple to resolve, we identify or construct a suitable context (that satisfies the targeted value of temporal features or locale features) in the existing program, and then transplant the malicious behavior (*i.e.,* resource feature) to the identified or constructed location. On the other hand, to alter the dependency features of the program, because dependency features such as external attributes usually require sophisticated ways (*i.e.,* method sequences) to achieve the specific desired control, we transplant the existing implementation of such control (*i.e.,* organ) from a donor app to the host app.

Such two-strategy design aims to simplify the existing software transplantation problem. In the first strategy, the transplantation is actually intra-app. We simply save and pass the unresolved dependency and contextual information (*e.g.,* values of parameters) in the program 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 [10]. Intra-app transplantation is feasible for temporal and locale features because synthesizing a new entrypoint or a new component within an existing Android program results in little or no impact to other areas of the program. Mutation of dependency features requires inter-app transplantation because synthesizing new dependencies within the program is challenging. The tight coupling of dependencies brings huge impact to other program behaviors and likely causes the mutated program to crash.

Note that although temporal features and local features all require the transplantation of the malicious behaviors, the donor (*i.e.,* area of code) that requires transplantation is different. The related code of a malicious behavior can be separated as the triggering part and the execution part. These two parts may not be in the same component. For example, in Figure 5, the malicious behavior of sending text message can be separated as the triggering part in the `OnCreate` method of 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-program transplantation, which are illustrated below.

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 3. 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 locale feature from Service to Activity.

B. Inter-method transplantation

Inter-method translation refers to the migration of malicious behaviors (*i.e.,* resource features) from a method to another method in the same component. We observe that such transplantation is commonly performed to mutate the temporal features. For example, the mutation of temporal feature in Listing 1 is inter-method transplantation (Lines 2-5 of `onClick` method in Listing 1 are transplanted from Lines 5-8 of the `onCreate` method in Figure 3b). In the case of temporal features, the organ that needs to be transplanted is the entry of the malicious behavior and its dependencies. 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 3b is the entry of the SMS sending behaviors. In order to locate the entry of the malicious behavior, we construct call graphs from the `entrypoint` of the program (corresponding to the feature to be mutated) to the malicious method call. We then mark the node directly connected to the entrypoint on the call graph as the entry of the malicious behavior.

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 program. For example, in Figure 5, 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.

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

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

Finally, we need to remove the organ from donor methods. If some of statements are dependent on 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. We 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 3b, we need to remove Line 7 while keeping other lines because Lines 9-13 are control-dependent on Lines 5-6.

C. 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 3c).

Inter-component transplantation follows the same process as inter-method transplantation except for two differences. First, in addition to temporal features, inter-component transplantation is also used to mutate locale features. As previously mentioned, to mutate local features, the organ to be transplanted is the execution part of the code. To extract such organ, we find the call graph node directly linked by the entrypoint of the execution part. Note that the entrypoint of the execution part can be different from the entrypoint of the malicious behavior. For example, in Figure 5, 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 inter-component communications to invoke the host method. To avoid crash caused by unmatching intent messages, the regnerator also adds conditional statements to avoid executing the existing code in the host method when such inter-component communications occur.

D. Inter-program transplantation

The inter-program transplantation is used to migrate the dependency feature of a malicious behavior in the donor app to the host app with identical malicious behavior. The extraction of the dependency feature is different from migration of the triggering/execution part of 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 the entrypoint of the program. We then migrate both parts of the organ into the host app.

VI. Testing on Mutated Apps

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

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

To address these challenges, we develop two techniques to assist the 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.

Listing 2: System logging of DougaLeaker before and after mutation

```
1 //Before
2 MainActivity start
3 onCreate() start
4 MyService start
5 onStartCommand() start
6 invoke sendTextMessage
```

```
1 //After
2 MainActivity start
3 onCreate() start
4 onClick() start
5 invoke sendTextMessage
```

Checking the robustness of mutated apps. We leverage random testing to check the robustness of a mutated app. In particular, we use Monkey [45], a random user-event-stream generator for Android, to generate UI test sequences for mutated apps. Each mutated app was tested against 5,000 events randomly generated by Monkey to ensure that the app does not crash [11] .

VII. Experiment

Implementation. We leverage Soot [55] as our underlying static analysis framework. We use Soot's submodule Dexpler [11] to convert Dalvik bytecode into the Jimple intermediate representation from which we perform feature extraction and feature mutation. We also leverage FlowDroid [31], a static taint analysis tool based on Soot, to provide a precise modeling of the Android component lifecycles and callback methods. Note that the current version of FlowDroid neither can handle native code nor can resolve reflective invocations in Java code when constructing the call graph; therefore, RTLD features in these cases cannot be extracted.

Dataset. Our subject set consists of a malware dataset and a benign app dataset. Our malware dataset starts with 3,000 malware randomly selected from Genome [66], Contagio [3], VirusShare [4], and Drebin [6]. 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 3240 apps in total. We implement the process of extracting RTLD features using third-party static analysis frameworks, including Soot [55] and FlowDroid [53]. To isolate and remove the effects of potential limitations of these frameworks, we run feature-extraction analysis on the complete subject set and remove any apps that cause a third-party tool to fail. The filtering gives us a final analyzable dataset of 1917 malware and 1935 benign apps. Given the large number of apps being excluded, we reassess the distribution of our final dataset. Our final malware dataset consists of 529 malware from Genome, 25 malware from Contagio, 287 malware from VirusShare, and 1076 malware from Drebin dataset. Our final benign app dataset 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 [6], [61] are performed on a server with Intel(R) Xeon(R) CPU 2.80GH with 38 processors and 80 GB of memory with a timeout of 80 minutes in total for processing 1917 malware and 1935 benign apps.

Baseline Approaches. We implement two baseline approaches for comparison with MRV: *random MRV* and OCTOPUS. We first develop a random transformation strategy (random MRV) to compare against confusion and evolution attacks. Instead of following the evolution rules and similarity metrics to 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 (Fig. 2).

We also implement a syntactic app obfuscation tool called OCTOPUS similar to DroidChameleon [49]. 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 (Sec. VI) after each obfuscation. If the testing passes, we then apply the next obfuscation to the obfuscated sample. If the testing fails, we restore the malware sample before the obfuscation. In our experiments, all semantic mutations including Random MRV and evolution/confusion attacks are performed after the syntactic obfuscation of OCTOPUS.

Malware detectors. We use a number of learning-based and signature-based malware detectors to evaluate the effectiveness of MRV. For learning-based malware detectors, we adopt AppContext [61] and Drebin [6]. **AppContext** leverages contextual features (*e.g.,* the events and conditions that cause the security-sensitive behaviors to occur) to identify malicious behaviors. In our experiments, AppContext generates around 400,000 behavior rows on our dataset (3852 apps), where each row is a 679-dimensional behavior vector. We conservatively label these behaviors (*i.e.,* marking a behavior as malicious only when the behavior is mentioned by existing malware diagnosis). The labeled behaviors are then used as training data to construct a Support Vector Machine (SVM) based classifier. **Drebin** uses eight features that reside either in the manifest file or in the disassembled code to capture the malware behaviors. Since Drebin is not open source, we develop our own version of Drebin according to its description [6]. Although Drebin extracts only eight features from a program, Drebin covers almost every possible combination of feature values resulting in a very large feature vector space. In fact, Drebin produces over 50,000 distinct feature values on our dataset (3852 apps). We perform ten-fold cross-validations to assess the effectiveness of AppContext and Drebin. Table II 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 [28] to pick the results of seven well-known anti-virus vendors (*i.e.,* AVG, Symantec, ESET, Dr. Web, Kaspersky, Trend Micro, and McAfee) and label an app as malicious if more than half of the seven suggest that the app is malicious. Following such procedure, only malware labeled as malicious are selected into our malware dataset, and thus all malware in our dataset can be detected by VirusTotal.

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

[11] Due to the limitation of the coverage of random testing, the mutated app passing the testing step can still be invalid. As future work, we plan to incorporate more intelligence-guided testing techniques [33], [60] in MRV testing.

TABLE II: Detection results of AppContext vs. Drebin on our dataset

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

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

TABLE III: Malware variants generated by different evasive techniques and undetected by VirusTotal

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

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

MRV, we produce both evasive and non-evasive variants and perform the detection after the variants are produced[12] (by skipping the detection step in Figure 2).

VIII. Result

A. Defeating existing anti-virus detection

Table III 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 and evolution attack. The row "Transformable malware" refers to the number of malware that can be mutated to a valid malware variant (*i.e.,* of all malware variants generated at different levels of an evasive technique, at least one of the malware variants can pass the testing). The row "Robust variants" shows the number of generated variants, and the last row shows the number of variants that can evade the detection of VirusTotal.

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

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

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

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

TABLE IV: Malware variants undetected (vs. all variants) by AppContext

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

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

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

B. Defeating learning-based detection

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

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

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

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

TABLE V: Malware variants undetected (vs. all variants) by Drebin

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

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

TABLE VI: 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

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

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

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

C. Effectiveness of attacks at each level

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

We have some interesting observations from Table VI. 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 evaluation (*i.e.,* temporal-locale-dependency) might not be the most optimal choice to evade some detectors. Ideally, we can explore different combinations of the mutation levels to maximize the number of undetected malware samples for each malware detector.

TABLE VII: Detection results of malware originally evading detection of AppContext or Drebin

Detector	A.+20%V.	A.+40%V.	A.+60%V.	A.+80%V.	A.+ V.
AppContext	171	165	152	140	123
Drebin	36	33	29	24	19

A. = All detected malware, V. = Variants

We also observe that most of unsuccessful variants produced at the dependency feature level are due to the fact that the malicious behavior cannot be triggered even 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 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.

D. Improving learning-based detection

We also perform a preliminary experiment to check whether the variants produced by MRV can improve the detection capability of existing techniques. Note that when using the original malware as training dataset, AppContext and Drebin fail to detect 178 and 38 malware samples correspondingly (Table II). We use these undetected malware samples as the testing dataset. As benign apps in the training dataset, we use the original 1935 benign apps. As malware in the training dataset, we combine previously detected samples with different portions (20%, 40%, 60%, 80%) of malware variants generated by MRV. As shown in Table VII, adding variants produced by MRV in the training dataset can help detect the malware samples that previously evade the detection.

E. Evolution patterns of RTLD features

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

Temporal features. We observe that lifecycle events are the temporal features that are most likely to evolve. In many malicious behaviors, the temporal feature evolves from a lifecycle event to a UI event, a system event, or another lifecycle event. Moreover, such evolution can be intra-component (*i.e.,* the entrypoint is 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 inter-component communications have been added to incorporate the evolution. Malware also frequently mutate temporal features for malicious behaviors triggered by system events. Except some system events commonly observed in benign apps (*e.g.,* `android.intent.action.PACKAGE_ADDED`, `android.intent.action.BOOT_COMPLETED`), most temporal features such as system events have evolved to

UI events or lifecycle events. The behaviors that were triggered by third-party ad-network intents (*e.g.,* `com.airpush.android.PushServiceStart`) also evolved[13].

Despite the frequent temporal-feature evolution, the evolution that mutates UI events to other temporal features seldom happens. Observing the overall evolution of several malware families, we find that the reason of such one-directional evolution is that to avoid being detected by users or dynamic app inspection, malware tend to decrease the frequency of triggering known malicious behaviors (by mutating the triggering event to UI event) while triggering new attacks more frequently. Such evolution also confuses the static detection. Because UI events are also commonly used in benign apps, the temporal feature resulted from the evolution makes malware less identifiable for detection techniques.

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

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

Composite Evolution. We have observed 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 leverages the system event `android.net.conn.CONNECTIVITY_CHANGE` to get notified when network connection changes so it can obtain the `NetworkInfo` from the system Intent message for new network connections. As the system event may be used as temporal features, the app evolves to leverage the system event `android.intent.action.USER_PRESENT`. Such event will get malware notified when a user is present after unlocking the screen. Then the malware starts a service and uses a timer in the service to repetitively invoke `getNetworkInfo` every 20 minutes. In this way, the malware author mutates the original temporal feature value (`CONNECTIVITY_CHANGE`) to a value (`USER_PRESENT`) completely unrelated to network connection. Meanwhile, the locale feature value is mutated from `receiver` to `service`, and adds the control dependency with timer. MRV can be especially powerful in detecting such sophisticated evolution.

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

IX. Discussion

Limitations of underlying infrastructures. The implementation of our MRV suffers from the limitations of standard static analysis in handling threading, alias, virtual methods, reflection, callbacks, and native. Also due to the limitations of Jimple's expressiveness along with the intermediate representation (IR) for program analysis tasks used in Soot (for the transplantation framework), the conversion from Dalvik code to Jimple may not preserve the complete program behavior. Such issue could be addressed by leveraging other more powerful tools such as dex2jar for program transformation. MRV mainly serves for testing static detection techniques. So MRV does not handle Java reflection or encrypted strings because static detection techniques usually do not handle these features. MRV does not aim to produce malware variants for all malware samples, but aims to produce a handful of malware variants to test and assess existing malware detection techniques.

Limitations of app testing. In testing processes for transformed apps, the testing checks only two main things: whether the program crashes or not and whether the malicious statements get invoked or not. Due to the unavailability of systematic functional test cases in regression tests, the testing cannot cover all the cases. Such issue could be addressed by manually creating regression tests for the malicious behavior. Additionally, it is infeasible to test whether malware become less profitable or easier to be detected by manual inspection.

Adware threat in experiments. In our experiments, we view one family of adware (*i.e.,* AdsDisplay) as one malware family due to the fact that both malware and adware are "unwanted software (UNP)", and differ significantly from benign apps according to a recent study [52].

Configuration of machine-learning-based classifiers. In practice, a machine-learning-based system typically has configuration parameters and thresholds. In our attack model, we do not focus on the determination of thresholds for decisions. We assume that the parameters are already searched for the optimal values via cross validation and prior knowledge.

Mitigation of MRV. The applicability of MRV largely depends on the effectiveness of malware features used in detection techniques to capture the tendency of the possible malware variation. Most existing malware detection techniques summarize patterns based on static and behavior analysis of

malware. Summarizing patterns without revealing the evolving tendency might not be sufficient to fight MRV. The evolving patterns of malware can be deemed as a concept drift in learning problems. Concept drift [47] is used as a generic term to describe the relation between the input and the target variable of a learning problem changes over time. Adversary activities can be one of the major sources to cause the change [68]. One way to cope with MRV is to adopt drift-aware adaptive learning algorithms [18], [26], [48] or concept drift detection [46] in the malware detection.

Coping with the malware evolution can also be deemed as a long-term dependency learning problem [14]. Not only we need to investigate recent malware, but also to study the gradient changes in a long period. Long Short Term Memory networks (LSTMs) [34] are a special kind of recurrent neural network designed for learning long-term dependencies. It would be an interesting direction to apply LSTMs on malware detection.

In addition to concept drift adaptation and neural networks, malware trend analysis [59] based on malware behaviors' temporal patterns [37], [54] is also highly desirable in assisting machine learning models to detect the evolving code structures and behaviors.

X. Related Work

Evasive techniques. Metamorphic malware [42], polymorphic malware [40] and other obfuscation techniques [24] have been developed to evade malware detection [63]. Semantic signature [23], behavior graphs [30] and other semantics aware techniques [56], [61], [9] have been developed to defeat against malware. To study how anti-malware products are resistant against transformed mobile apps, Droidchameleon [49] is developed as a systematic framework with various transformation techniques for mobile app study and they found the transformed apps can easily evade detection. Unlike MRV, DroidChameleon only performs syntactic obfuscation, which can be easily be detected by semantic-based detection tool such as AppContext. PraGUARD [44] performs assets encryption on malware samples to assess the role of external resources (i.e., assets) in the detection for the anti-malware tools PraGUARD focuses on assessing detection of external resources, while MRV focuses on assessing detection of apps behaviors. Two approaches focus on different problems, complementary to each other. Replacement attack [57] is proposed to poison behavior-based specifications by concealing similar behaviors of malware variants. Replacement attack can impede malware clustering [13]. MalGene [38] is developed to automatically locate evasive behavior in system call sequences and therefore extract evasion signatures. Different from these works, MRV can evade both anti-virus tools and machine learning based classifiers.

Recent evaluations [52] on ML-based malware detection techniques suggest that more feature number does not necessarily improve the performances due to the non-informative features [6] and noisy features. Recent study on PDFrate [50] evaluates existing learning-based malware detection techniques in different evasion scenarios. Xu et al. [58] propose an evading technique based on classification score feedback that can manipulate PDF malware samples to evade detection of PDFrate and Hidost. Carmony et al. [17] manipulate JavaScript payload in PDF malware to evade detections. The major difference between these works and MRV is that MRV requires much less knowledge about machine learning model to launch the attack. Moreover, prior work mainly focuses on pdf malware while MRV automatically generates malware variants for android apps.

Adversarial machine learning. Adversarial machine learning [15], [16], [36], [50] studies the effectiveness of machine learning techniques against an adversarial opponent. Adversaries frequently attempt to break many of the assumptions that practitioners make (*e.g.,* data has various weak stochastic properties; data do not follow a stationary data distribution). Generally, the adversary is assumed to have full (or partial) information related to three components of learner: learning algorithm, feature space and training dataset. In our work, we propose a targeted attack assuming a malware developer knows the information about the feature space and classifier.

Program transplantation. We leverage program transplantation technique for program transformation. Given an entry point in the donor and a target implantation point in the host program, the goal of automated program transplantation [10], [43], [53] is to identify and extract an organ, all code associated with the feature of interest, then transform it to be compatible with the name space and context of its target site in the host. In this work, we broaden program transplantation concept to intra-app transplantation that needs to deal with the side-effect of the removal of the organ (compared to inter-app transplantation), where effective heuristics are proposed to automatically identify the target implantation point in the host program.

Contextual-based malware detection. To identify potential malware among rapidly increasing number of published apps, existing researches leverage both dynamic and static analysis. Pegasus [19] constructs permission event graphs using static analysis to model the effects of the event system and API semantics, and performs model checking to enforce the policies specified by users. AppIntent [62] presents a sequence of GUI events that lead to data transmissions and lets analysts decide whether the data transmissions are intended. AsDroid [35] detects stealthy app behaviors by identifying mismatches between API invocations and the text displayed in the GUIs. DroidSIFT [65] builds the weighted contextual API dependency graph to fight against malware, variants and zero-day attacks. MassVet [22] determines whether a new app is malicious or not by finding the commonalities between the new one and the known malware based on features generated from UI structure and method control-flow graphs. Triggerscope [29] seeks to automatically identify logic bombs in Android malware using symbolic execution, path predicate reconstruction and minimization, and interprocedural control-dependency analysis to enable the precise detection and characterization of detecting triggers. BareCloud [39] is developed to detect evasive malware based on bare-metal dynamic malware analysis by comparing malware behavior observed in the transparent bare-metal system with other popular malware analysis systems including emulation-based and virtualization-based analysis systems. Because all these works leverage contextual/non-essential features the RTLD feature model can be used to generate malware variants to evade the detection of these techniques.

XI. Conclusion

In this paper, we propose two attacks that mutate malware variants to evade detection. The core idea is changing the behaviors that the malware detector concerns. To achieve this goal, we conduct static analysis, phylogenetic analysis, machine learning, and program transplantation to systematically

produce the new malware mutations. To the best of our knowledge, it is the first effort towards solving the malware-evasion problem using malware feature confusions and evolutions without *any* knowledge of the underlying detection models. MRV opens up intriguing, valuable venues of future MRV applications. First, proposed attacks can be used to evaluate the robustness of malware detectors and therefore quantify the differentiability of particular features. Second, MRV can help discover the potential attack surface so MRV can be further used to assist iterative design of malware detectors. Finally, the program transplantation framework capable of changing malware features and behaviors can be written as malicious payload that resides in malware itself so as to create adaptive malware.

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*.

[21] 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 *24th USENIX Security Symposium (USENIX Security 15)*, pages 659–674, 2015.

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

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

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

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

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

[27] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

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

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

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

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

[32] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, 2013.

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

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

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

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

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

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

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

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

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

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

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

[44] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

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

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

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

[48] M. Pechenizkiy. Predictive analytics on evolving data streams anticipating and adapting to changes in known and unknown contexts. In *High*

Performance Computing & Simulation (HPCS), 2015 International Conference on, pages 658–659. IEEE, 2015.

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

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

[51] 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 *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

[64] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *CCS*, 2014.

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

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

[67] Z. Zhu and T. Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 767–778. ACM, 2016.

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