Mystique: Evolving Android Malware for Auditing Anti-Malware Tools

Guozhu Meng*, Yinxing Xue*, Chandramohan Mahinthan*, Annamalai Narayanan*, Yang Liu*, Jie Zhang* and Tieming Chen†
*School of Computer Engineering, Nanyang Technological University, Singapore
†Department of Computer Science and Technology, Zhejiang University Of Technology, China

ABSTRACT

In the arms race of attackers and defenders, the defense is usually more challenging than the attack due to the unpredicted vulnerabilities and newly emerging attacks every day. Currently, most of existing malware detection solutions are individually proposed to address certain types of attacks or certain evasion techniques. Thus, it is desired to conduct a systematic investigation and evaluation of anti-malware solutions and tools based on different attacks and evasion techniques. In this paper, we first propose a meta model for Android malware to capture the common attack features and evasion features in the malware. Based on this model, we develop a framework, MYSTIQUE, to automatically generate malware covering four attack features and two evasion features, by adopting the software product line engineering approach. With the help of MYSTIQUE, we conduct experiments to 1) understand Android malware and the associated attack features as well as evasion techniques; 2) evaluate and compare the 57 off-the-shelf anti-malware tools, 9 academic solutions and 4 App market vetting processes in terms of accuracy in detecting attack features and capability in addressing evasion. Last but not least, we provide a benchmark of Android malware with proper labeling of contained attack and evasion features.

Keywords

Android Feature Model, Defense Capability, Malware Generation, Evolutionary Algorithm

1. INTRODUCTION

Malware detection is always one of the central topics in cybersecurity. Anti-malware tools (AMTs) are getting more advanced, but as a result surviving malware is getting increasingly sophisticated. Generally speaking, the development of AMTs usually lags behind the advance of new malware, since new *malware variants* (similar malware generated by software obfuscation or configuration techniques) and zero-day *vulnerabilities* (a weakness that allows an attacker to exploit the target system) keep emerging every day.

In retrospect, Android malware undergoes a stunningly rapid increase in a short time of last five years. In 2010, we witness the industrial age of mobile malware, and also in that year Geinimi was one of the first found malware that attacked the Android platform and used the infected phone as part of a mobile botnet [3]. Ever since then, a larger number of sophisticated mobile malware is created by attackers due to the prevalence of Android phones. With regard to the defence side, traditional approaches relying on textual or hexadecimal signatures [46] are incapable of detecting variants of existing malware and new ones. To catch up the trend, in research community, machine learning based approaches [5, 7, 9, 52] and information-flow analysis based approaches [8, 27, 34] are proposed to detect obfuscated malware and their variants. Recently, new attacks (transformation attacks [41, 42] and collusion attacks [28, 46]) are revealed, and they fail the existing detection approaches. Thus the similar arms race between malware and anti-malware is unexceptionally observed on Android platform.

Recently, Android malware exhibits a variety of attack behaviors, including leaking user privacy, escalating privilege without permission, conducting unknown financial charge, and abusing application functionality. In real malware, these attack behaviors may coexist in order to increase the damage and success probability of the attack. Even worse, evasion techniques (e.g., multiple-level obfuscation [41, 42]) are further applied on the code or deployment package to evade the scanning of AMTs. Hence, the combination of different attack behaviors and evasion techniques indeed exist in real-world Android malware [55] — such a fact hinders the understanding of the reason why AMTs fail in detection.

The main challenge in auditing the AMTs is the lack of the evaluation criteria and well labeled benchmark such that we cannot evaluate and the strength and weakness of AMTs systematically. To our best knowledge, existing AMT evaluation is based on benchmarks like GENOME [55] and DREBIN [7], which classify malware based on families only. GENOME and DREBIN are not suitable for auditing AMTs due to three reasons: 1) the malware are old and well recorded in malware repository of AV tools, 2) there is no a comprehensive coverage of different attacking and evasion techniques, 3) there is no index (or label) to different malware attributes (e.g., attack behaviors, obfuscation techniques and anti-debugging techniques), except for (inaccurate) family names. By far, only one existing work [41] discusses the resilience of different *Defence strategys* (DS) against obfuscation techniques.

In this paper, we propose to separate different attack behaviors and evasion techniques into basic reusable features — to summarize the attack features and evasion features of malware on Android. Here, *attack feature* (AF) means malicious behavior of a certain attack, which links to implementation of the functional requirements (intention) of malware. *Evasion feature* (EF) means the ability of malware to evade the scanning of AMTs, including a variety of code complication and transformation techniques that change no functional requirements of malware. In this way, we can develop a meta model for Android malware by modularizing various attack

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

DOI: http://dx.doi.org/10.1145/2897845.2897856

features into the various *building blocks*. Hence this meta model allows us to generate malware variants to cover different AFs and EFs, and evaluate how different DSs react to each individual feature as well as their different combinations.

Technically, we start with detecting and analyzing the malicious code among the similar yet different malware variants in an Android malware sample. We then modularize the malicious code from different malware families into different AFs, and identify evasion techniques into EFs. Once features are modularized, we adopt the concept of Software Product Line Engineering (SPLE) and build a feature-oriented architecture [31] as the malware meta model to capture the different features and their constraints inside malware.

With the malware meta model, we apply a multiple-objective evolutionary algorithm (MOEA) to mimic the evolution of the malware. MOEA performs *gene crossover* (i.e., exchanging features of two samples) and *mutations* (i.e., selecting or deselecting feature under mutation) on the current malware generation to produce next malware generation. To guide the evolution to generate *better* malware, we define the fitness function for selecting next generation by maximizing the number of attack behaviors, minimizing of evasion techniques needed and the expected detection rate.

Finally, we develop the proposed malware generation process into a tool called MYSTIQUE, and use it to audit 57 off-the-shelf anti-malware tools and 9 academic solutions in terms of detection ratio and capability with 10,000 generated malware. With the experiment results, we test four commonsense hypothesis of AMTs. To check the capability of online vetting of app stores, we upload 12 generated malicious apps onto 4 mainstream app stores. In most cases, our malware passes the online vetting process. Furthermore, we propose some possible enhancements for the existing malware detection approaches.

To sum up, we make the following contributions:

- We recognize the Android malware as AFs and EFs and present them in a meta model. We consider and maintain the traceability between the features and their corresponding code in MYSTIQUE.
- Based on the meta model, we develop an SPL architecture to generate new Android malware by an MOEA. Our approach is implemented in an automated framework named MYSTIQUE.
- We survey and evaluate the state-of-the-art AMTs using our generated malware. The experiments show that the existing AMTs are quite weak at detecting these new malware — a detection ratio of less than 30% on average. We propose the countermeasures in order to detect the malware generated by MYSTIQUE.
- We have generated over 10,000 samples of Android malware by combining different attack and evasion features. They can serve as a benchmark to assess detection capabilities of AMTs, as they cover different representative combinations of features, which are missing in the current malware benchmarks.

2. BACKGROUND

2.1 Software Product Line Engineering

Software product line engineering (SPLE) is a paradigm of developing a set of similar software. SPLE adopts feature-oriented domain analysis [31] for requirement analysis and builds core asset architecture for reuse [17]. Here, *features* are attributes of system and requirements of end-users. Technically, SPLE is a two-phase approach composed of domain engineering and application engineering. The task of domain engineering is to build the software product line (SPL) architecture consisting of a common program base and various variant features, while the application engineering focuses on derivation of new products by different customizations of variant features applied onto the common program base.

The concept of *feature model* in domain engineering is to represent the features within the product family as well as the structural and semantic (*require* or *exclude*) relationships between those features [31]. Since the proposal of SPL, feature model has even been characterized as "the greatest contribution of domain engineering to software engineering" [18].

A feature model is a tree-like hierarchy of features. The structural and semantic relationships between a super (or compound) feature and its subfeatures can be specified as:

- *Mandatory* – A mandatory feature must be selected if its super feature is selected,
- *Optional* – An optional feature is optional to be selected,
- *Or* – If the super feature is selected, at least one of the subfeatures must be selected,
- *Alternative* – If the super feature is selected, exactly one among the exclusive subfeatures should be selected.

Besides the above tree-structure constraints (TCs) between features, cross-tree constraints (CTCs) are also often adopted to represent the mutual relationship for features across the feature model. There are three types of common CTCs:

- f_1 *requires* f_2 – The inclusion of feature f_1 implies the inclusion of feature f_2 in the same product.
- f_1 *excludes* f_2 – The inclusion of feature f_1 implies the exclusion of feature f_2 in the same product, and vice versa.
- f_1 *iff* f_2 – The inclusion of feature f_1 implies the inclusion of feature f_2 in the same product, and vice versa.

In SPLE, a feature model (e.g., that of Linux kernel [48]) may even contain thousands of features. Selecting an optional set of features, which satisfies the constraints (i.e., TCs and CTCs) among features and attains the optimisation of product attributes, is not a trivial problem. Basically, selecting such an optional set is a searching problem, to address which, in SPLE community, the multiple objective evolutionary algorithms (MOEAs) are commonly adopted. In SPLE, the desired products have a set of attributes (e.g., performance, costs and defects) to be optimized, which means multi-objective optimization is needed in product design when competing features exist. IBEA (Indicator-Based Evolutionary Algorithm), has been proved to the best MOEA for solving the problem of optimal feature selection [44, 50].

2.2 Overview of Android Attacks

Application cloning has been observed in Android application market by industrial developers and the academic community [13, 54]. In GENOME [55] and DREBIN [7], many malware samples share the common attack behaviors. We identify five types of threats from GENOME malware, namely privacy leakage, privilege escalation, financial charge, abuse of functionality, and ransomware. In this study, we focus on the modeling and generation of the malware of privacy leakage. The rationale is that this type of malware constitutes 78.7% in GENOME, and most of existing academic prototypes (e.g., FLOWDROID [8], DROIDSAFE [27]) are proposed to address privacy leakage.

Privacy Leakage. A damage of this type of attack is the possibility of exposing users' sensitive information such as account credentials, preferences and contacts. In Android, specifically, sensitive information that can be leaked is twofold — contacts, messages, personal information available on social networks, and financial information directly accessible by malicious users. These are examples of *explicit* privacy, which is mentioned in [47]. Another kind of privacy is *implicit* privacy. Implicit privacy denotes the information that malicious users cannot directly use — the attacker has to analyze it in order to reveal valuable information. For example, Schlegel

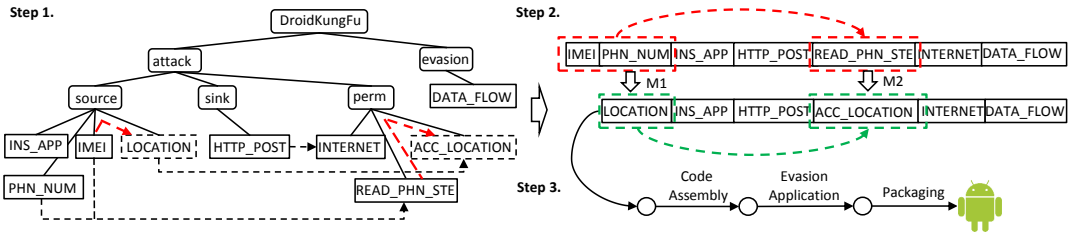

Figure 1: A running example to illustrate the generation of new variant of DroidKungFu

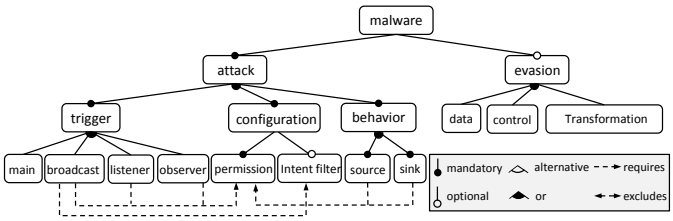

Figure 2: The partial feature model of privacy leakage malware

et al. [45] present an approach that can gather audio data from on-board sensors and use it to recognize commercial credentials.

As introduced above, the same type of attacks may have various implementations or adopt different channels to conduct the attacks. We regard these different implementations of the same type of attacks as *attack features*. For example, the attack features of privacy leakage include a mandatory information source, an optional flow of information, and a mandatory sink of information. The implementation of accessing information source may refer to some finer-grained attack features, e.g., getting the phone number by calling method getLine1Number. Details and more examples can be found in Section 3 and Fig. 1. To sum up, various attack features serve as the basic building blocks for constructing malware.

In Section 7, we evaluate the state-of-the-art AMTs, including academic prototypes and anti-virus tools, with our generated malware of privacy leakage. Note that our approach works for all types of Android malware, as long as attack and evasion features of a certain attack type are identified and modularized.

2.3 Adopting SPLE for Malware Generation

Identifying common attack features among Android malware enables feature-oriented domain analysis (FODA) [31], which is a domain analysis method to decouple and model software systems for better reuse and reconstruction. FODA adopts the feature model (§ 2.1) to analyze the commonality and variability inside a software family, and guide the variant generation for the family.

Malicious apps in an Android malware family usually share much of source code. To better understand the behaviors of Android malware, we perform the feature-oriented domain analysis on each malware family to identify and model their attack features (§ 4.1). Once the attack features are identified and modularized, the state-of-the-art development paradigm, SPLE, naturally comes into the picture for large-scale and flexible malware generation.

With the generated malware via SPLE, we can systematically evaluate how and to what extent AMTs and the associated evidences can help detect different attack or evasion features. Beyond that, we can audit these AMTs and their detection mechanisms behind.

3. MYSTIQUE OVERVIEW

This section explains the high-level idea of our approach MYSTIQUE and depicts the major steps of our approach with a running example. In addition, we present the potential challenges in this study.

3.1 Mystique Overview

We show the overview of the malware generation using Fig. 3. The input of MYSTIQUE includes the original malware collection (i.e., GENOME [55] in this work), and the identified meta model with the code snippets of features. The output is a collection of generated malware samples that are labeled with features inside.

Here, we briefly describe the work flow. First, the attack features (AF) and evasion features (EF) used in the original malware collection should be identified and modularised (§ 4.1). Attack features are the requirements of malware from the prospective of attackers, and atomic functionalities from the prospective of SPL. They are the fundamental composition of activities (i.e., behaviors) of apps. After finishing the feature-oriented domain analysis (FODA) for the 49 malware families in GENOME, we remove the duplicated AFs and add the remaining AFs in the feature model shown in Fig. 2. Meanwhile, the corresponding code of these AFs are also modularised in separated methods or classes, in such a way that the architecture of our malware product line is built up. EFs are mainly from evasion techniques in flow and code. EFs in flow include multiple ways from source to sink, such as Lifecycle [8] and ICC [38], and EFs in code are inspired by the transformation of code [41, 42]. Once the features in our FODA are available and the corresponding feature code are modularized, we adopt the Indicator-Based Evolutionary Algorithm [44, 50] to generate malware with the three objectives: *aggressiveness*, *evasiveness* and *detectability* (§ 5).

Running Example We take *DroidKungFu* in GENOME as an example. *DroidKungFu* belongs to malware of privacy leakage, which steals sensitive information (e.g., IMEI code, phone number). Fig. 1 shows the feature model for malicious behaviors contained in *DroidKungFu*. The feature model consists of necessary features for launching attacks as well as their relationship in between. For example, *DroidKungFu* obtains IMEI code and phone number, which belong to the feature Source. Feature Source has 4 optional sub-features, among which feature IMEI and PHN_NUM require permission feature READ_PHN_STE that relates to permission (§ 4.1).

After obtaining the feature model, we encode the attack features (malicious behaviors) as chromosome in gene. Specifically, each bit of the chromosome represents the existence of an attack feature — 1 for existent and 0 for non-existent. The *crossover* operation and *mutation* operations are performed on the chromosome during the evolution. In step 2 of Fig. 1, we mutate features IMEI and PHN_NUM to feature LOCATION. According to feature dependency, the original permission feature READ_PHN_STE for IMEI is not required any more, while feature LOCATION acquires permission feature ACC_LOCATION. After that, the gene of a new malware variant is produced.

In step 3 of Fig. 1, the code of each feature in gene will be assembled. Details on assembling triggers and manifest file are in Section 5. In addition, to audit the capabilities of AMTs, we also employ obfuscation to evade the detection, then one malware variant is created and can be used for auditing AMTs. Owing to the gene of the generated malware variant, we can evaluate what feature combinations can evade what defence strategies.

3.2 Technical Challenge

To generate the sound and workable malware, we face the following technical challenges:

- **C1: Construction of feature model.** A variety of features are contained in malware, and used in different ways. It is difficult to identify and extract the representative features in malware. Section 4.1 addresses this challenge.
- **C2: Malware measurement.** With the feature model (i.e., the meta-model of malware), we still need some goals to guide automated malware generation. Thus, we define three objectives: aggressiveness, evasiveness and detectability for measuring malware (§ 5). Since these objectives are competing (e.g., highly offensive malware is generally more detectable), we use a MOEA to select feature from feature model in Section 5.
- **C3: Validation of generated malware.** As the malware is generated according to the feature model, we want to prove their maliciousness — whether malicious behaviors can be triggered and carried out on real Android devices. To address this challenge, in Section 7, we conduct the following experiments. For the attack of privacy leakage, we set up a dummy server or device to receive the sensitive information sent from malware.

4. FEATURE-ORIENTED DOMAIN ANALYSIS OF ANDROID MALWARE

We identify the common building blocks for Android malware as *attack features* and *evasion features*, and propose to use feature model as the meta model to capture the malware. Different from focusing on requirements in malware ontology analysis [37], we consider and maintain the traceability between the features and their corresponding code in our model. One partial feature model of malware of privacy leakage in GENOME is shown in Fig. 2[1]. Note that feature-oriented domain analysis relies much on the domain knowledge of security experts, and only feature and their code relevant to attacks are manually modularized. We totally identify 266 attack features and 14 evasion features. Note that our feature model is not a complete one, but can be extended to covering new attack behaviors and evasion methods.

4.1 Attack Feature

Attack features refer to features that are generally relevant to the malicious behaviors of a certain type of attack. They can be further categorized into the following three types:

Trigger Feature. *Trigger* defines the entry points for malicious attack behaviors. Triggers are roughly categorized into GUI-based and non GUI-based [51, 52]. Since non GUI-based triggers are not easy to be discovered by users, thereby more suspicious, we only consider non GUI-based triggers without the interaction with end users. There are four kinds of non GUI-based trigger features are mainly identified in GENOME: main, broadcast, listener and observer. The trigger main denotes that malicious behaviors are triggered from the startup; malicious behaviors can be triggered from a broadcast message by registering a BroadcastReceiver; malware can also register a listener to listen the changes on states (e.g., location); malware can register an observer on a ContentProvider to listen to its changes.

Configuration Feature. Two kinds of configuration features are relevant to malicious attack behaviors in malware: *permission* and *intent filter*. Android provides a permission-based mechanism to avoid the abuse of system sensitive operations, e.g., invoking sensitive Android APIs. Many malicious behaviors in malware require certain permissions to attain attack goals. For example, it needs the permission android.permission.READ_PHONE_STATE to obtain the IMEI code of the device. Feature Intent_Filter defines the acceptable Intent that can be captured by Android components. For example, if one BroadcastReceiver is assigned with intent filter android.provider.Telephony.SMS_RECEIVED, it can capture broadcast messages indicating an incoming SMS.

Behavior Feature. Attack trigger and configuration features are all assistant to the core attack features — behavior features. In an attack of privacy leakage, there are mainly two types of features: Source and Sink. Source is responsible for stealing sensitive information of device, and sink is responsible for sending out sensitive information. Based on the manual domain analysis of GENOME, 11 kinds of source features and 2 kinds of sink features are identified [1].

Note that the partial feature model in Fig. 2 mainly illustrates the high-level organization of these features. Each leaf feature in Fig. 2 may have several subfeatures, e.g., feature Source has 11 variant sub-features in an *Or* relationship, and each variant feature may also have several implementation features (modularized code) in an *Alternative* relationship. Interested readers can refer to Section 5 and our tool website [1] for more details.

4.2 Evasion Feature

Information flows from *source* to *sink* in privacy leakage attack can be obfuscated in three different ways as follows:

Control based Evasion. Malicious behaviors can be obfuscated by complicating the control flow of the attack. Android provides an amount of callback functions to guarantee implicit control flow. Besides, each component in Android has its own lifecycle together with a set of built-in APIs for lifecycle management. In an attack of privacy leakage, the control flow usually involves the interactions between different components in Android. Thus, it is feasible for the attacker to hide the malicious code into the different stages of components and trigger it under certain scenarios. For example, each Android component has a life cycle, the method invocation sequence of which is defined in the framework layer of Android. A certain method will be invoked if the component is in a specific state. In Inter-Component Communication (ICC), there is also one mechanism for implicit control flow if an Intent object is not assigned with a determined class [27, 38].

Data based Evasion. Attacks like privacy leakage must conduct data transmission. Such transmission can happen between different methods, classes, apps, or even different channels (i.e., external persistent storage and memory).

- **Persistent Storage**. On Android, applications may exchange data through persistent storage. There are three types of persistent storage provided by Android: *file, shared preferences and SQLite database*. They can be used for applications or components to exchange data — they provide an implicit data flow from one component to another.
- **Memory**. Data that is temporarily stored in a specific memory location (e.g., an object in the Java heap) might be accessed globally. As a consequence, once there is a component or method to fetch data from that memory location, it establishes a data flow from where the data is stored to where it is fetched.

Both. Attacks can be further complicated by combining the obfuscations on both control and data flow. As Intents are the main vehicle for app communication, they can be used for a purpose of advanced obfuscation. One intent can be either explicit or implicit. Explicit intents have a specific class to start, while implicit intents do not specify the corresponding class, and the system will select the most well-suited class or application to execute. An explicit intent can only invoke a specific component, which is defined in

[1]The complete feature model of Android malware is available in our website [1].

the constructor, or by calling setComponent(ComponentName) or setClass(Context, Class); an implicit intent can be received by many well-suited components. It appoints potential receivers by setting an action in the constructor or setAction(String) (Meanwhile, it can be instrumented with a data type to restrict its receivers). In addition, an Intent object can be bundled with some data by invoking putExtra, which generates a data flow from the caller component to the callee component. Therefore, Intent can influence the execution order of the app and also the data flow if enclosed with *extras*.

In addition, we use the transformation attacks [41, 42] to complicate and transform the source code at the implementation level.

Transformation Attacks. We have selected 12 types of transformation attacks, such as identifier renaming, data encryption, code reordering, to obfuscate the generated malware. Different from the previous evasion techniques, the transformations cannot change the behaviors or flows from source to sink. It is a kind of non-behavior evasion technique since the transformations only change the lexical information or code structure. However, by adding a lot of noise to the previous code, it may bypass the detection of AMTs which consider code structure or lexical information. With the reordering of code, it may break some AMTs based on static analysis.

5. MULTI-OBJECTIVE GUIDED MALWARE GENERATION

This section is devoted to the Android malware generation process in MYSTIQUE. As shown in Fig. 3, MYSTIQUE takes a malware feature model as the input and generates various malware variants. During the process, each malware is encoded as a DNA sequence based on their values for AF and EF in the feature model. The malware evolution is an iterative process in accordance with the principle of survival of the fittest, where we randomly choose an initial population of malware satisfying the input feature model, and select better malware in the each generation afterwards based on the fitness function, i.e., the malware measurement function in our case. The four steps in each iteration of malware evolution are as follows.

- **Step 1: Feature Selection.** Given the previous generation n of malware, step 1 is to produce new malware by applying crossover and mutation operations on malware from generation n. Considering the three objectives, the IBEA is used to choose the fittest ones in each iteration.
- **Step 2: Code Assembly.** To make the malware run on a real device, MYSTIQUE conducts to validate the assembled code and package it into a deployable app. It includes the setup of *triggers* that act as entry points of malicious behaviors, the configuration of manifest file, and malware packaging.
- **Step 3: Evasion Application.** After evasion features are selected, the corresponding evasion techniques are applied. Note that the evasion is based on the source code, without changing the malicious intent or behaviors of the constructed malware.
- **Step 4: Objective Evaluation.** In this step, we calculate the objective functions and choose the fittest for the next iteration. We also need to check whether the evolution is finished due to convergence of EA or reaching the upper limit of iteration times.

The selected features of a feature model is encoded using an array-based chromosome as shown in the step 2 of Fig. 1. Given a chromosome of length n, array indices are numbered from 0 to $n-1$. Each feature (no matter AF or EF) is assigned with an array index starting from 0. Each value on the chromosome is z_i such that $z_i \in \mathbb{Z} \wedge z_i \in \{0,1\}$, where 0 (resp. 1) represents the absence (resp. presence) of the feature. Given a feature model M, we define a function $f_M : Fea(M) \rightarrow \{\mathbb{Z}, \perp\}$ that maps each feature f of the feature model M to an array index. $f_M(f_1) = \perp$ denotes that there is no array index that is assigned for the feature f_1. Similarly, we define ${f_M}^{-1} : \mathbb{Z} \rightarrow Fea(M)$ as a function that maps a given array index to the feature it represents. Thus, gene crossover is just the array exchange at a certain index, and gene mutation is bit flipping of the value at a certain index of the array.

To serve as the goals of malware generation, we propose three objective functions in the evolution of malware: *aggressiveness*, *evasiveness* and *detectability*. As the results of the arms race, malware are getting more aggressive with minimum evasion features needed, but less detectable.

Given a chromosome x, we represent it as a bit vector of attack and evasion features, where $\{f_1^a ... f_n^a\}$ denotes the set of n attack features and $\{f_1^e ... f_m^e\}$ denotes the set of m evasion features. The objective functions are defined as follows:

Definition 1 ***Aggressiveness*** *means the severity of damages that malware may cause to users, which is measured by the number of contained AFs and formally defined as*

$$\mathscr{F}_1(x) = \sum_{i=1}^{n} \|f_i^a\| \tag{1}$$

where $\|f_i^a\|$ returns 1 if f_i^a is selected and returns 0 if not.

Definition 2 ***Evasiveness*** *means the efforts to hide the malicious intent and evade the detection. Attackers want to minimize such effects of using evasion techniques to evade detection. It is measured by the number of contained EFs and defined as follows*

$$\mathscr{F}_2(x) = \sum_{i=1}^{m} \|f_i^e\| \tag{2}$$

where $\|f_i^e\|$ returns 1 if f_i^e is selected and returns 0 if not.

Given a chromosome x and the set of AMTs S_d, $\{d_1 ... d_t\}$, introduced in Section 6, we have the following definition for the last objective function:

Definition 3 ***Detectability*** *means the difficulty in detecting the malware. It can be measured by detection results of AMTs and defined as follows*

$$\mathscr{F}_3(x) = (\sum_{i=1}^{|S_d|} \mathscr{D}_i(x)) / \mid S_d \mid \tag{3}$$

where $\mathscr{D}_i(x)$ returns 1 if the malware of x is detected by the tool d_i and $\mid S_d \mid$ denotes the number of the tools.

5.1 Feature Selection via IBEA

Rather than encode three objectives into one weighted fitness function, we treat all the three objectives equally and solve *Multi-objective Optimization Problems (MOPs)* using the Pareto dominance relation [30].

A k-objective optimization problem could be written in the following form[2] (in our case, $k = 3$):

$$Minimize\ \vec{\mathscr{F}} = (\mathscr{F}_1(x), \mathscr{F}_2(x), ..., \mathscr{F}_k(x)) \tag{4}$$

where $\vec{\mathscr{F}}$ is a k-dimensional objective vector and $\mathscr{F}_i(x)$ is the value of $\vec{\mathscr{F}}$ for ith objective.

Definition 4 *Given two chromosomes $\vec{x}, \vec{y} \in B^n$ and an objective vector $\vec{\mathscr{F}} : B^n \rightarrow R^k$, $\vec{x}$* ***dominates*** *$\vec{y}$ ($\vec{x} \prec \vec{y}$) if*

$$\forall i \in \{1, ..., k\} \quad \mathscr{F}_i(\vec{x}) \leqslant \mathscr{F}_i(\vec{y}) \tag{5}$$

$$\exists j \in \{1, ..., k\} \quad \mathscr{F}_j(\vec{x}) < \mathscr{F}_j(\vec{y}) \tag{6}$$

otherwise $\vec{x} \not\prec \vec{y}$

[2]Evasiveness and detectability need to be minimized, but aggressiveness needs to be maximized. In implementation, maximizing is minimizing the negative value of the objective.

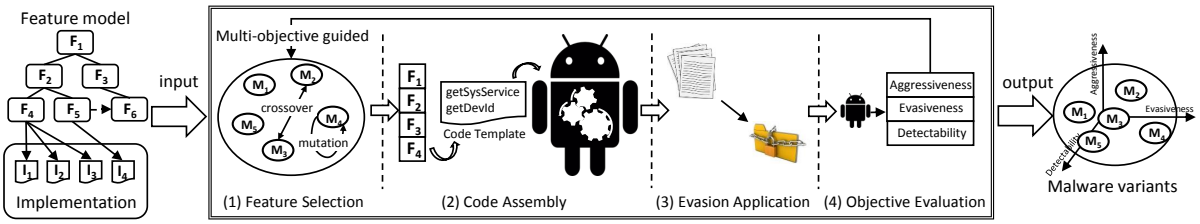

Figure 3: Multi-objective guided malware generation

Algorithm 1: Multi-objective guided malware generation

Input: *featureModel*: the feature model of Android malware, *maxIter*: the maximum number of iterations of the generation process, *popSize*: the size for each generation
Output: *allMal*: a list of malicious apps with different feature combinations

```
1  allMal ← ∅;
2  for 1 to maxIter do
3      newGeneration ← ∅;
4      if allMal == ∅ then
5          for 1 to popSize do
6              malware ← randomFeatureSelection(featureModel);
7              newGeneration ← newGeneration ∪ {malware};
8      else
9          for newGeneration.size() < popSize do
10             select i,j ∈ [1,|allMal|];
11             malware = ibea_crossover(allMal[i],allMal[j]);
12             newGeneration ← newGeneration ∪ {malware};
13             select k ∈ [1,|allMal|];
14             malware = ibea_mutation(allMal[k]);
15             newGeneration ← newGeneration ∪ {malware};
16     for mal ∈ newGeneration do
17         evaluate(mal);
18     for mal_i ∈ newGeneration do
19         if ∃mal_j | mal_j ∈ newGeneration ∧ i ≠ j • mal_j ≺ mal_i then
20             newGeneration ← newGeneration \ mal_i ;
21     if newGeneration ⊂ allMal then
22         break;
23     allMal ← allMal ∪ newGeneration;
24 return allMal;
```

Definition 5 *Given chromosomes $\vec{x}$ and a set of chromosomes $S_{\vec{x}}$, $\vec{x}$ is* ***non-dominated*** *iff*

$$\forall \vec{x}_i \in S_{\vec{x}} \quad \vec{x}_i \not\prec \vec{x} \tag{7}$$

Algorithm 1 shows how IBEA guides the feature selection. The input of this algorithm is the feature model of Android malware, the number of iterations for the generation process, and the population size. In the beginning, it creates the initial population, randomly selecting features in the feature model (line 4 to 7), otherwise we can generate new malware derived from the existing malware (line 9 to 15). First, it selects two candidates with a probability and do an *ibea_crossover* operation (line 11) to generate a new malicious app. Second, it selects one candidate in a probability and do an *ibea_mutation* operation (line 14) to generate a new malicious app. Once the generation is created, all apps in this generation are evaluated by the proposed three objectives to get the fitness value (line 16, 17). The algorithm utilizes the Pareto dominance relation to remove malware that is dominated by others in the evaluation (line 18 to 20). The algorithm will stop once the selected features converges (line 21) or exceeds the maximal times of iteration (line 2).

5.2 Code Assembly

In this section, we elaborate how to assemble code according to the selected features. The input of this step is a list of features. The output of this step is the assembled source code of the malware.

The translation from selected features to corresponding source code is inspired by Aspect-Oriented Programming (AOP) [33]. One feature has at least one implementation. For example, the source feature TELEPHONY::IMEI can be implemented by invoking the Android APIs—getSystemService, getDeviceId, in sequence. All the implementations are integrated into a candidate app, and we select specific implementations in terms of the selected features to generate a malicious app. Meanwhile, in the code generation, we also write scripts to automatically check constraints from feature dependencies and context, and generate the configuration files, such as *AndroidManifest.xml*.

Constraints in Assembly. To assure the soundness and validity of the assembled code, we have the following rules:

- **Feature dependency constraint**. Feature model has features dependencies inside, which are constraints to be satisfied in code assembly. For example, the permission feature android.permission.READ_PHONE_STATE is needed for feature getDeviceId. When a component attempts to start an activity via ICC, in order to maintain the activity list in the history stack, it has to set the flag FLAG_ACTIVITY_NEW_TASK.
- **Context constraint**. We consider extra constraints to be verified in code assembly, specifically, some operations can only occur in a specific context. For example, the incoming SMS message is only accessed in the context of onReceive, and the receipt of ICC in a service is onStartCommand. All these constraints are not feature-relevant at requirement level, but specific to implementation. All these constraints should be satisfied to avoid runtime exception.

5.3 Evasion Application

For information leakage, AFs and EFs are orthogonally separated, which implies the independence between the choice of AFs and EFs. After AFs and EFs are selected by IBEA, the EFs are categorized into two types: flow based ones or transformation based ones.

Evasion based on source-sink flow. The evasion is applied in constructing the code, and the purpose is to complicate the flow between the source and the sink. One malicious behavior may stretch through multiple components. Therefore, we employ the evasion features (§ 4.2) to obfuscate flows. For example, Android Lifecycle is used to complicate control flows and ICC can be used to complicate both control and data flows.

Evasion based on transformation. This evasion is applied after the step of code assembly. DROIDCHAMELEON can directly work with the deployment package of Android app. For the 12 transformations mentioned (§ 4.2), we provide 12 EFs. If EFs are selected by IBEA in step 1, we will later apply the corresponding transformations.

5.4 Objective Evaluation

According to Definition 1-3, we calculate the fitness value for each generated malicious app. The fitness value is used as the guidance to the feature selection in the next generation. We inspect all apps in the new generation. If no new feature combination is produced, the evolution process converges and would be terminated

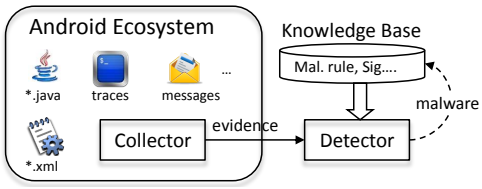

Figure 4: The working process of an AMT

as line 21 in Algorithm 1. Finally, we get the collection of new generated malware that serves the benchmark for auditing AMTs.

6. BRIEF ON ANTI-MALWARE TOOLS

In this section, we explore the capabilities of off-the-shelf AMTs. Fig. 4 shows the working process of a typical AMT. Basically, it contains *Collector* which collects evidence (§ 6.1.1) from Android ecosystem. The evidence is sent to a *Detector* which determines whether the app is malicious or not. Generally, the detector makes the decision based on *Knowledge Base* (§ 6.1.2). Additionally (as shown by dashed line), the confirmed malware can be supplemented into the knowledge base to update the knowledge base. Finally, according to our survey and testing on AMTs, we list detection mechanisms adopted by mainstream AMTs in Section 6.2.

6.1 Detection Mechanism

6.1.1 Evidence Collection

AMTs need to collect and extract evidence as proof to identify Android malware. Evidence can be collected from various sources in Android ecosystem, ranging from different program entities (e.g., *AndroidManifest.xml* and class files) to various program output of dynamic execution.

The source of evidence are threefold: *manifest file*, *binary code* and *runtime information*. In the manifest file, AMTs can collect as evidence the information: *used permissions* [7, 21, 43], *hardware components* [7], *Android components* [7], and *Intent filters* [7]; AMTs can extract from binary code *Android APIs* [7, 9, 25, 27], *constant strings* [7, 51], *control flow* [6, 23, 27, 43], *data flow* [20, 23, 27], and *program dependency graph* [52]; Some AMTs may run apps in a real device or an emulator to collect the *execution traces* of Android apps [11]. As malware often launches attacks via HTTP or SMS, [19] collects the HTTP behaviors to fingerprint Android apps. Interested users can refer to [1] for more details.

As the attack of privacy leakage involves both data and operations, some attack hints can be found in all the above types of evidence. Some other types of attack are more closely relevant to some certain types of evidence, e.g., the attack of functionality abuse is mainly behavior-oriented and highly identifiable based on **AA** and **AB**.

6.1.2 Knowledge-base Detection

The knowledge base is the basis and criteria of distinguishing malware from benignware. After obtaining the evidence, AMTs that use the knowledge base confirm malware by the following strategies:
Signature of Known Malware (SKM). An abundance of malware collections [2, 7, 55] is publicly available for industrial and academic research. Such abundance of malware enables to fast detect malware based on signatures or features. For example, DROIDSIFT extracts *behavior graph* from known malware. Many commercial anti-virus tools use the hash value of malware as the signature [22]. However. approaches based on exact matching with known malware (e.g., comparing hash value) are not resistent to new variants.
Attack Pattern (AP). To overcome the limitation of signature based detection, existing AMTs can leverage the knowledge of attacks, including attack targets, attack techniques, attack camouflages and so forth. We herein call it *attack pattern* [14, 20] — generally, some form of modeling of attack behaviors with aforementioned evidence.

6.2 Anti-malware Tools & Hypothesis

According to the general analysis techniques employed in detection, AMTs can be categorized into the following four types. Some tools, such as [9, 25, 26, 52] that use both of static analysis and machine learning, are categorized into machine learning approach.
Machine Learning. Owing to the popularity and availability, we mainly analyze and audit the following machine learning approaches: DREBIN [7], ADAGIO [26], ALLIX *et al.* [6] and REVEALDROID [25].
State Analysis. Considering the efficiency and ease of setting up, most of existing approaches rely on static analysis. Since we construct malware of privacy leakage, we consider the following open-source detection tools targeting privacy leakage: SCANDROID [24], FLOWDROID [8], DROIDSAFE [27] and ICCTA [34].
Dynamic Analysis. Approaches based on dynamic analysis can be accurate at rumtime, but they face the difficulty in triggering malware and the scalability issue. Besides, they are hard to set up. We only audit the famous tool TAINTDROID [20] on this side.
Anti-virus Tools. We use the online service of VIRUSTOTAL [4] to audit the state-of-the-art Anti-virus tools. VIRUSTOTAL provides the recent version of 57 anti-virus tools.

In this study, we focus on evaluation of different detection mechanisms rather than comparison of the particular tools. The above list of AMTs may not be complete, but we try to cover the existing detection mechanisms. Before auditing AMTs, We propose the following four commonsense hypothesis :

Hypothesis 1 *Mainstream AV tools, which rely on signature or pattern based approaches, cannot detect the variants of the existing malware, even those with similar attack features.*

Hypothesis 2 *Evasion features in privacy leakage, e.g., flow complication, can help the malware to evade the detection, despite which detection approach is used by the AMTs.*

Hypothesis 3 *AMTs based on dynamic analysis should be more accurate than those based on static analysis or machine learning, regardless of the time and the difficulty in setup.*

Hypothesis 4 *The human check of malware in online app store, involving both static and dynamic analysis, is the most complete and sound solution to detect malware in reality.*

7. MALWARE AND AMT EVALUATION

MYSTIQUE is implemented in about 12K lines of Java code. Moreover, test scripts written for experiments are of 1K lines of Shell and Python. All the experiments are conducted on a Ubuntu 14.04 machine with Intel Xeon(R) CPU E5-16500 and 16G memory. In this section, our experiments are aimed to answer the following research questions:

RQ1. Are the modularized AFs and EFs valid? Is the generated malware valid and workable?

RQ2. Can we use the generated malware to audit AMT? Are the Hypothesis 1 to 4 in Section 6.2 accepted or rejected?

RQ3. Is our generated malware representative? How useful is MYSTIQUE in generating malware?

7.1 Evaluation Subjects

To evaluate the effectiveness of MYSTIQUE and the defense capabilities of AMTs, we generate multiple sets of malicious apps for different evaluation targets with MYSTIQUE. The malware is grouped based on its attack targets, and covers multiple attack and

evasion features. On the other hand, we use the malware to test the defense capabilities of AMTs, especially, the state-of-the-art public AMTs introduced in Section 6.2. The evaluation subjects are described in the following two aspects.

Offence: to evaluate the strength of the malware generated using MYSTIQUE. Each malware sample has at least one attack target, which is listed in Section 4.1. We give feature labels for malware to assess the attack capabilities. All the features used in MYSTIQUE feature model are manually summarized from the 1,260 malware samples in GENOME. Totally, we have 266 attack features and 14 evasion features in our feature model (§ 4.1). We sketch a diagram in Fig. 5 of the cumulative distribution for each kind of AF defined in GENOME. Since EFs are difficult to be categorized from the code, we do not show the distribution of EFs.

Defense:. to evaluate the four types of tools (§ 6.2) to cover a complete protection from three aspects: *untrusted app analysis*, *install-time checking*, and *continuous runtime monitoring* [49]. We need an initialization for machine learning and dynamic analysis tools. For machine learning tools, we select all 1,260 malware samples in GENOME, and 1,260 benign apps from Google Play as their training set. For dynamic analysis tools, we implement a driver in Python to simulate all possible triggers in our scope, e.g., starting an app, receiving an SMS message, changing the geography location. Interested readers can refer to the trigger list in [1].

7.2 RQ1: Validity of Generated Malware

We validate the generated malware from two aspects.

Proof of Program Synthesis. We assure that the flows of privacy leakage in malware are logically true. In detail, we verify the two phases of the automated malware generation: **p1**, feature selection (§ 5.1), and **p2**, transformation from feature model to code (§ 5.2).

- **Proof of p1**. In the process of feature selection, we select appropriate candidate features, conforming to the constraints in the feature model. It guarantees there are sufficient and necessary features to construct malware.
- **Proof of p2**. Constraints on the unique runtime environment of Android (§ 5.2) should be satisfied. For example, consuming operations in Android apps cannot be executed in the main thread, and hence we have to create a child thread to execute consuming operations. In code assembly, we write scripts to make sure all the implementation constraints are satisfied.

To sum up, for given features, this step is to assure that all the requirement and implementation constraints are satisfied.

Malware App Validation. The last step is to valid the final malware app to test whether it can leak privacy information. To this end, we set the target URL and phone number to our *honeypot* that the information would be sent to. We use the running example to illustrative the validation process. We generate a malicious app using the features of malware in the running example (Fig. 1). The selected features are as follows:

```
Triggers-MAIN::STARTUP
Sources-TELEPHONY::IMEI, TELEPHONEY::PHONE_NUMBER
Sinks-HTTP::APACHE_POST
(dependencies) PERMISSION::READ_PHONE_STATE
```

We set the target URL to our *honeypot* web site, in which there is a responding web page written in PHP to store the received message from the generated malware. Since there are 30 types of sources in the feature model, we use MYSTIQUE to generate 30 malicious apps accordingly, each of which contains one kind of sources. For simplicity, we construct one flow that satisfies the constraints defined in the feature model for the privacy leakage, by selecting one satisfiable trigger and sink, and setting up the acquired permissions. We execute them on a physical Android device. Our honeypot successfully collects all sensitive information sent by these malicious apps.

Table 1: Detection ratio of privacy leakage malware in GENOME

Malware Family	# Samples	Detection Ratio (%)			
		DA	SA	ML	AV
DroidKungFu3	309	0	53.7	100	70.2
AnserverBot	187	0	51.0	99.7	74.7
BaseBridge	122	0	60.7	99.2	73.0
DroidKungFu4	96	0	10.2	100	70.3
Geinimi	69	0	40.1	99.3	71.9
Pjapps	58	0	45.6	98.9	71.6
KMin	52	0	37.8	100	72.0
GoldDream	47	0	55.6	100	70.4
DroidKungFu1	34	0	60.2	100	75.6
DroidKungFu2	30	0	67.0	100	73.7

7.3 RQ2: Auditing of AMTs

In this section, we aim to evaluate the AMT using the generated malware and test the four hypothesis.

First, we test the deployed AMTs on GENOME malware as the baseline understanding of AMTs. Note that we only choose malware with privacy leakage attack, which contains 78% of the 1,260 samples in GENOME. The results are presented in Table 1, where machine learning tools and anti-virus tools perform well in detecting existing malware. As the dataset GENOME originated from 2010, anti-virus tools (AVTs), which are mainly based on signature and pattern matching, can accurately detect the malware with a recall of 71.9% on average. There are still some AVTs that perform poorly, e.g., Bkav (0%), CMC (0%), Malwarebytes (0%) and TheHacker (0%). Since machine learning tools use 60% of malware samples in GENOME as the training set and the remaining 40% as the testing set, they outperform the other tools with a higher recall.

Static analysis and dynamic analysis are more time-consuming compared to the previous two approaches, due to the program analysis they conduct. Static analysis tools has yet achieved around 48.4% of detection ratio of GENOME malware. For the dynamic tool TAINTDROID, it fails to detect existing malware in GENOME. The problem is attributed to the limited support of TAINTDROID to source or sink types, and the compatibility issues when running out-of-date malware in latest Android OS.

Second, we use MYSTIQUE to generate 100 generations of malware without evasion features to evaluate the detection ratio (DR) of AMTs. Then we add evasion features into the malicious apps to re-evaluate the DR. As shown in Table 2, there are two columns for each kind of AMTs, of which the first column is the DR **without** evasion features, and the second column "(E)" is the DR **with** evasion features. All the values of DRs are calculated as the average values amongst tools of a specific type. We summarize the hypothesis testing results as follows.

H1. The Susceptibility of AVs to Unknown Malware. Mainstream AVs employ signature- or feature-based approaches. The detection capabilities depend on the completeness and timeliness of malware database, and also the abstraction of malware. Generally, they perform very well in detecting known malware as in GENOME experiment above: they achieve a 71.9% recall on average, and 27 (out of 57) AVs can even detect at least 99% of malware samples in the experiment. However, they perform poorly in detecting our generated 0day malware. According to the detection results of our generated malware, only 18 generated malware samples can be detected by the union of these AVs. For example, ESET-NOD32 detects 3 malware samples as "a variant of Android/TrojanSMS.Agent.BLY". By further inspection, we find that the 3 samples steal the SMS messages. Specifically, they share one common behavior: it mon-

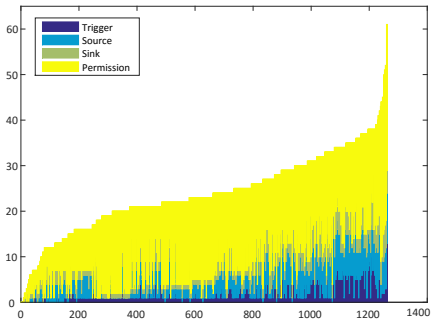

Figure 5: Cumulative AFs in GENOME samples

Type	Feature	DR (%)
Source	TELEPHONY::SIM_SERIAL	21.1
Source	TELEPHONY::SIM_COUNTRY	12.5
Trigger	BROADCAST::android.bluetooth.device.action.NAME_CHANGED	12.5
Trigger	BROADCAST::android.intent.action.ACTION_SHUTDOWN	12.5
Trigger	BROADCAST::android.intent.action.PACKAGE_REMOVED	12.5
Source	SMS::INCOMING_SMS	1.6
Source	BUILD::SDK_INT	1.6
Trigger	BROADCAST::android.provider.Telephony.SMS_RECEIVED	1.6
Trigger	BROADCAST::android.intent.action.PACKAGE_RESTARTED	1.6
Sink	HTTP::SOCKET_GET	1.6

Figure 6: The significance of attack features in detection

Table 2: The objective value of generated malware during evolution

Gen	#Vars	AFs				#EFs	Detection Ratio (%)							
		#Triggers	#Sources	#Sinks	#Perms		DA	DA(E)	SA	SA(E)	ML	ML(E)	AV	AV(E)
10	50	35.9	13.4	4.6	74.9	5.6	17.2	13.4	32.5	12.5	42.5	41.7	0.0	0.0
20	50	31.8	7.6	4.5	82.6	7.5	34.2	14.3	25.0	13.5	25.0	22.5	0.0	0.0
30	50	33.8	9.8	3.1	75.2	4.8	24.5	14.3	27.5	15.0	20.0	20.0	0.0	0.0
40	50	29.5	9.9	2.4	78.3	5.1	14.1	14.1	17.5	15.9	32.5	29.5	0.0	0.0
50	50	32.9	8.2	2.4	81.9	8.0	22.0	15.9	17.5	0.0	27.5	25.0	0.0	0.0
60	50	31.2	10.5	2.2	80.5	7.5	21.0	10.5	17.5	12.8	27.5	22.5	0.0	0.0
70	50	27.6	10.5	2.4	74.5	6.2	6.7	4.8	17.5	15.0	25.0	22.5	0.0	0.0
80	50	28.5	10.3	4.8	70.3	3.1	19.5	14.2	18.2	12.5	27.3	25.0	0.0	0.0
90	50	33.3	9.0	2.9	77.5	6.6	5.6	5.6	10.0	5.8	25.0	21.0	0.0	0.0
100	50	36.9	10.0	4.0	76.5	5.4	10.2	8.7	10.0	6.0	22.3	20.0	0.0	0.0

itors the change of the Content Provider of SMS, steals all SMS messages, and sends out to a specific remote server.

We can conclude that AVs have made efforts to infer the semantics of code as the behavior is split into two methods. However, the inference is quite limited. We crafted malware samples by employing evasion techniques, which cannot be detected any more. In general, we consider H1 is **accepted**.

H2. The Insignificant Impact of Evasion Techniques. We have generated two malware datasets, one of which contains malware samples without any evasion features, and the other contains malware samples with arbitrary evasion features. From the comparison of detection results, evasion features rarely effect the detection results of AVs. It can help to evade the detection of dynamic and static analysis (43.7% of reduction in DR). Since the dynamic analysis tool TAINTDROID tracks the flow of information in the system, it fails to detect the privacy leakage once the flow is complicated by involving ICC or implicit data flow. The static analysis tools that perform a code analysis from the source to sink, can overcome complicated transformation attacks and behavior-level evasion techniques. For example, ICCTA takes into the account ICCs during different components of apps, can identify behaviors of privacy leakage occurring across multiple components. However, static analysis in ICCTA still has some flaws. It cannot track the data flow across persistent storage, such as file, SQLite or shared preferences. Static analysis tools usually employ API-matching to identify sources and sinks. Therefore, they can be easily defeated by involving dynamic loading techniques, such as reflection, constant encryption. Moreover, for machine learning based tools, evasion features have a little impact on DR, which is not significant enough (the differences of ML and ML(E) in Table 2 are within 5%). We observe that the higher #EFs does not necessarily lead to a lower DR.

Thus, we consider that H2 is **partially accepted** — certain evasion can only work for certain detection approaches and too many evasions may not better bypass the detection.

H3. Diverse Detection Capabilities of AMTs. Based on the detection results to our malware benchmark, we test H3 by evaluating the weakness and strength of each type of approaches.

- Dynamic analysis is a kind of black box testing, which focuses on the input and output of sensitive information to apps, while they do not consider how the behavior is implemented. Therefore, the detection capabilities depend on the coverage of sources, sinks and the communication channels between. Our experiments show that TAINTDROID can track sensitive information obtained from specific Android APIs, such as getDeviceId and getLine1Number. It does not track the information from incoming SMS message and Content Provider, etc. It performs well for the communication channel ICC and file-based channel. However, SQLite and shared preferences can help bypass its detection.
- Static analysis is more scalable than dynamic analysis. However, it lacks of information during runtime and thereby its capabilities are limited. Nowadays, there are some works [36] using *symbolic execution* to mitigate the lacking of runtime information.
- We compare the detection results of two malware sets, one of which has more attack features and the other has less attack features. The dataset with more attack features are more likely to be detected, while machine learning based approaches are susceptible to malware with less attack features. Another comparison occur between two tools REVEALDROID and DREBIN. Although DREBIN has considered more features, its detection ratio is improved a lot. Therefore, the significance, rather than the number, of features can better facilitate the detection. As shown in Fig. 6, we list five attack features which are easiest to be detected, and five attack features which are hardest to be detected. Interested readers please refer to [1] for a complete list of significance of attack features in the feature model.
- It is reasonable for AVs to use a fast approach with a low false positive rate. Our observation is that AVs mainly aim at detecting known malware. Hence, AVs work in a reactive way, not in a proactive way.

To sum up, we consider that H3 should be **rejected**. Considering the detection results of TAINTDROID in Table 2, we cannot confirm that dynamic tools can produce high detection accuracy, although they can provide more accurate information in detection. The problem lies in the difficulty in triggering malicious behaviors in execution. Note that due to the unavailability of other dynamic tools, we cannot generalize our conclusion for all dynamic tools.

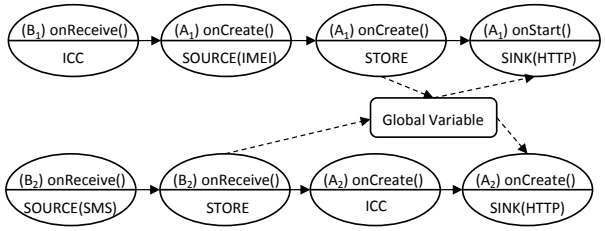

Figure 7: Malicious behaviors to be repackaged

Table 3: The capabilities of vetting process in modern marketplaces

#Benign Base	Google Play	GetJar	SlideMe	TorrApk
1	✓	✗	✓	✗
2	✗	✗	✗	✗
3	✗	✗	✗	✗

H4. Strong Vetting Process in Modern App Stores. Modern Android app stores employ multiple techniques to inspect the submitted apps and protect their marketplaces. Google Play has turned from an offline dynamic analysis-*Bouncer* [35] to a manual check by human experts [32]. Currently, Android app stores *GetJar*[3], *SlideMe*[4] and *TorrApk*[5] all inspect the submitted apps by human experts.

Since our generated malware has no normal functionalities other than malicious behaviors, it got rejected when we submit it into these four app stores. To address this, we download three open-source benignware, which have been verified by AMTs and approved by Google Play. We inject our malicious behaviors into their source code, repackage them and then submit them to the four Android app stores. One example of malicious behaviors is shown in Fig. 7. And it acquires 5 permission and steals SMS messages and identity information of device into a particular server.

For each of the 3 benignware (benign base), we select 4 malware samples from our benchmark and inject them into the benign base. Here are the 4 malware samples: 1) one malicious app without evasion features; 2) one malicious app is generated by adding evasion features into the first app; 3) an optimal malware sample in our benchmark. 4) a random choosen one from our malware benchmark. The 4 different malicious apps from the same benign base is submitted to the four different app stores. Table 3 shows the detection ratio of these apps by AMTs, ✗ and ✓ indicate an app is approved (not detected) and rejected (detected) by the corresponding app stores, respectively. From this experiment, we can conclude that the vetting process of Android app stores still have severe flaws, and can be easily bypassed. Although human inspection can judge the quality of apps of high confidence, the security of apps is not fully inspected.

According to our observations, we consider that H4 should be **rejected**. Note that the malware samples that are used for injection are with a ratio of 0% to 22.8% to be detected. Thus, the vetting results are not significantly better than the results of our AMTs. We suspect that the vetting process also uses the AMTs for detection.

7.4 RQ3: Representative Malware and Usefulness of Mystique

In this section, we conduct a controlled experiment to assess the effectiveness of MYSTIQUE to obtain optimal malware from the attacker's view. The basic idea is to use a small set of AFs and EFs for fast convergence, and evaluate the resulting malware when the evolution stops.

The experiment is conducted as follows: 1) pick up 10 samples of malware which can be detected. 2) use IBEA algorithm to generate new variants by combining or mutating features in the initial population of malware. 3) stop if no more optimal malware is generated.

The 10 samples are from malware family DroidKungFu3, AnserverBot, BaseBridge, DroidKungFu4, Geinimi, Pjapp, KMin, GoldDream, DroidKungFu1 and DroidKungFu2 as shown in Table 1. The extracted features are as follows. In addition, we consider all 14 types of evasion features in this experiment.

Triggers:
- [T_1] STARTUP,
- [T_2] android.intent.action.BOOT_COMPLETED,
- [T_3] android.intent.action.BATTERY_CHANGED,
- [T_4] android.intent.action.NEW_OUTGOING_CALL
- [T_5] android.provider.Telephony.SMS_RECEIVED,

Source:
- [SU_1] PACKAGE::INSTALLED_APK,
- [SU_2] SMS::ALL,
- [SU_3] SMS::INCOMING_SMS,
- [SU_4] TELEPHONY::IMEI,
- [SU_5] TELEPHONY::IMSI,
- [SU_6] TELEPHONY::PHONE_NUMBER,
- [SU_7] TELEPHONY::SIM_SERIAL

Sinks:
- [SI_1] HTTP::APACHE_GET,
- [SI_2] HTTP::APACHE_POST,
- [SI_3] HTTP::SOCKET_POST,
- [SI_4] SMS::SEND_TEXT_MESSAGE

Permissions:
- [P_1] android.permission.INTERNET
- [P_2] android.permission.PROCESS_OUTGOING_CALLS
- [P_3] android.permission.RECEIVE_BOOT_COMPLETED
- [P_4] android.permission.READ_PHONE_STATE
- [P_5] android.permission.RECEIVE_SMS
- [P_6] android.permission.SEND_SMS

Evasion:
- [E_1] Control based evasion
- [E_2] Data based evasion
- [E_3] Transformation attacks (12 types of transformation)

Initially, MYSTIQUE selects features randomly to construct 10 malware samples as the initial population. MYSTIQUE evolves based on the fitness value of newly-generated malware. After 30 iterations, MYSTIQUE obtains the optimal malware of which the fitness values reach optimum in three objectives. The optimal malware contains 16 attack features and 3 evasion features. AFs in the optimal malware are {T_1, T_3, T_5, SU_1, SU_2, SU_4, SU_5, SU_7, SI_1, SI_2, SI_3, SI_4, P_1, P_2, P_3, P_6}, and EFs contains control based evasion, data based evasion and one transformation. We put the optimal malware and more details on our tool website [1] for public observation.

8. DISCUSSION

8.1 Threats to Validity

The internal threats for experiment results are from three aspects. First, we mainly consider privacy leakage attack and their behaviors. However, the logic of this attack is quite straightforward and we have the limited number of AFs. Second, for EFs, we mainly take into account the flow complication and transformation attacks. Last, we use default parameters and set-up of IBEA for malware evolution. Further investigation should be conducted to see the effects of different parameters and set-up.

The external threats mainly stem from the choice of malware samples for FODA. Currently, our malware samples for FODA are only from GENOME. As GENOME originated from 2010, it may contain lots of out-of-date malware. To ensure the timeliness of malware, we need to further investigate what malicious behaviors are contained in other malware collections. Another threat is due to the public availability of the AMTs that we can audit. Especially,

[3]http://developer.getjar.mobi/
[4]http://slideme.org/
[5]https://www.torrapk.com/

for tools based on dynamic analysis, the source code is required for better debugging and testing the malware. At this moment, we only have the source code of TAINTDROID. In future, we plan to audit more AMTs based on dynamic analysis.

8.2 Countermeasure for Generated Malware

According to our experiments on generated malicious apps and the detection results of AMTs, we present three suggestions for future research on Android malware detection.

- **A Refined Source&Sink Pattern**. Generally, the recognition of sources and sinks is the first step for static- and dynamic-analysis tools. Consequently, they need to track the flow of information (obtained by sources) in either the program or the runtime environment. However, most of existing works [8, 20, 27, 34] identify sources and sinks by doing a matching with Android APIs, such as SUSI [40]. However, there exist some sources which cannot be represented as APIs. For example, the number of incoming calls can be obtained from the context of ⟨PhoneStateListener⟩onCallStateChanged. Although SUSI includes the methods getLatitude and getLongitude as sources, malware can use the method toString instead to fetch the specific latitude and longitude to bypass AMTs' tracking, and **these kinds of sources exist in our benchmark**. Hence, one refined pattern for sources and sinks facilitate the detection of privacy leakage.
- **Full Consideration of Communication Channels**. There exist many communication channels in Android, through which information is transmitted. Besides ICC provided by Android, malicious apps can communicate via system memory or persistent storage. In addition, there emerge side channel attacks in Android [15]. All of these advanced techniques hinder the detection of malware. Therefore, modern detection approaches should follow the development of attacks firmly and supplement domain knowledge from time to time.
- **Correct Understanding of Malicious Behaviors**. Current approaches based on machine learning lack an understanding of essences of malicious behaviors. Features extracted from apps are usually separated or not directly relevant to malicious behaviors. Although machine learning tools achieve 91.4% on accuracy in the training, they can only detect less than 9.5% of generated malware in reality (§ 7.3). Therefore, with a tolerable loss of efficiency, machine learning based approaches can learn the essences of malicious behaviors deeply to increase their performance on Android apps in the wild. For example, they can employ static analysis to extract the relationship between different features [25, 51, 52].

9. RELATED WORK

Android Malware Generation Aydogan and Sen [10] propose an approach to generate Android malware with a genetic algorithm. The newly generated malware originate from the crossover and mutation of malware in GENOME [55], and they conducted experiments to show that the new malware variants can easily bypass the detection of anti-virus tools. Cani *et. al.* [12] employ μGP to automatically create new malware which is undetectable for anti-virus tools, and injects it into a benign app to construct a Trojan horse.

Malware Evasion Techniques. Christodorescu et al. [16] firstly give a formal definition for obfuscation, and these techniques can be used by hackers to modify their malware to evade the detection of anti-virus tools and analysis of security analysts. In order to hinder dynamic analysis of Android malware, Petsas et al. [39] proposes three heuristics to check if malware is running on an emulated device or a real device, thereby decide whether to execute malicious behaviors. The three heuristics contains-static heuristics, such as IMEI code, routing table; dynamic heuristics, such as sensor data, and; hypervisor heuristics, such as QEMU scheduling.

Anti-malware Auditing. Christodorescu and Jha [16] leverage four types of obfuscation techniques to test the capabilities of commercial anti-virus tools. In addition, they propose an algorithm to extract the unique signature by which anti-virus tools use to identify malware. ADAM [53] is an automatic and extensible platform to test and audit Android anti-virus tools. It employs several transformation techniques to generate polymorphic malware, and test 10 prestigious anti-virus tools. DROIDCHAMELEON [41, 42] collects three types of transformation attacks in Android, and the authors have used these attacks to audit the off-the-shelf detection tools. Huang *et al.* [29] assess the detection capabilities of 30 top anti-virus tools from two aspects: malware scanning and engine updating. They reveal hazards of evasion in malware scanning, and null-protection windows during the update of engine.

10. CONCLUSION

We propose a feature model to describe the behaviors in malware for the ease of understanding and detection. We present MYSTIQUE, an Android malware generation framework to automatically generate malware with specific features. The generated malware is used to explore the agressivity of attack features, and efficiency of evasion techniques. We provide 10,000 generated malicious apps which can be used to evaluate the emerging AMTs and thereby help to enhance the security of Android ecosystem.

Acknowledgments

This work is supported by the National Research Foundation, Singapore under its National Cybersecurity R&D Program (Award NRF2014NCR-NCR001-30). This work is also sponsored by the National Science Foundation of China (No. 61572349, 61272106).

References

[1] Mystique | Evolving Android Malware for Auditing Anti-Malware Tools. https://sites.google.com/site/malwareevolution/.
[2] VirusShare. http://www.virusshare.com.
[3] 10 Years of Mobile Malware Whitepaper. http://www.fortinet.com/sites/default/files/whitepapers/10-Years-of-Mobile-Malware-Whitepaper.pdf, 2014.
[4] VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com, 2015.
[5] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *SecureComm*, 2013.
[6] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. Machine Learning-Based Malware Detection for Android Applications: History Matters! Technical Report 978-2-87971-132-4, 2014.
[7] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.
[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, pages 259–269, 2014.
[9] V. Avdiienko, K. Kuznetsov, A. Gorla, and A. Zeller. Mining Apps for Abnormal Usage of Sensitive Data. In *ICSE*, 2015.
[10] E. Aydogan and S. Sen. Automatic Generation of Mobile Malwares Using Genetic Programming. In *Applications of Evolutionary Computation*, volume 9028, 2015.
[11] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based Malware Detection System for Android. In *SPSM*, pages 15–26, 2011.

[12] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda. Towards Automated Malware Creation: Code Generation and Code Integration. In *SAC*, pages 157–160, 2014.

[13] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *ICSE*, pages 175–186, 2014.

[14] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *NDSS*, 2013.

[15] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security*, pages 1037–1052, 2014.

[16] M. Christodorescu and S. Jha. Testing Malware Detectors. In *ISSTA*, pages 34–44, 2004.

[17] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition, Aug. 2001.

[18] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Ttools and Applications*. Addison-Wesley, 2000.

[19] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In *IEEE INFOCOM*, pages 809–817, 2013.

[20] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, pages 1–6, 2010.

[21] W. Enck, M. Ongtang, and P. D. McDaniel. On Lightweight Mobile Phone Application Certification. In *CCS*, pages 235–245, 2009.

[22] Essam Al Daoud and Iqbal H. Jebril and Belal Zaqaibeh. Computer Virus Strategies and Detection Methods. 1(2), 2008.

[23] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *FSE*, pages 576–587, 2014.

[24] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Checking Interation-Based Declassification Policies for Android Using Symbolic Execution. Technical report, 2009.

[25] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek. Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware. Technical Report GMU-CS-TR-2015-10, 2015.

[26] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *AISec*, pages 45–54, 2013.

[27] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[28] H. Gunadi and A. Tiu. Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System. *CoRR*, abs/1311.2362, 2013.

[29] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu. Towards Discovering and Understanding Unexpected Hazards in Tailoring Antivirus Software for Android. In *AsiaCCS*, pages 7–18, 2015.

[30] H. Ishibuchi, N. Tsukamoto, and Y. Nojima. Evolutionary Many-Objective Optimization: A Short Review. In *CEC*, pages 2419–2426, 2008.

[31] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Nov 1990.

[32] E. Kim. Creating Better User Experiences on Google Play. http://android-developers.blogspot.ro/2015/03/creating-better-user-experiences-on.html, 2015.

[33] R. Laddad. *AspectJ in Action, Second Edition*. 2009.

[34] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[35] H. Lockheimer. Android and Security - Official Google Mobile Blog. http://googlemobile.blogspot.sg/2012/02/android-and-security.html, 2012.

[36] K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson. Checking Interation-Based Declassification Policies for Android Using Symbolic Execution. Technical Report arXiv:1504.03711v2, 2015.

[37] D. A. Mundie and D. M. McIntire. An Ontology for Malware Analysis. In *ARES*, pages 556–558, 2013.

[38] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *USENIX Security*, pages 543–558, 2013.

[39] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *EuroSec*, pages 5:1–5:6, 2014.

[40] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS*, 2014.

[41] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *AsiaCCS*, pages 329–334, 2013.

[42] V. Rastogi, Y. Chen, and X. Jiang. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.

[43] J. Sahs and L. Khan. A Machine Learning Approach to Android Malware Detection. In *EISIC*, pages 141–147, 2012.

[44] A. S. Sayyad, T. Menzies, and H. Ammar. On the Value of User Preferences in Search-based Software Engineering: A Case Study in Software Product Lines. In *ICSE*, pages 492–501, 2013.

[45] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*, Feb. 2011.

[46] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*, 2011.

[47] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yüksel, S. A. Camtepe, and S. Albayrak. Static Analysis of Executables for Collaborative Malware Detection on Android. In *ICC*, pages 631–635, 2009.

[48] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse Engineering Feature Models. In *ICSE*, pages 461–470, 2011.

[49] D. J. J. T. SUFATRIO, T.-W. CHUA, and V. L. L. THING. Securing Android: A Survey, Taxonomy, and Challenges, May 2015.

[50] T. H. Tan, Y. Xue, M. Chen, J. Sun, Y. Liu, and J. S. Dong. Optimizing Selection of Competing Features via Feedback-directed Evolutionary Algorithms. In *ISSTA*, pages 246–256, 2015.

[51] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating Malicious and Benign Mobile App Behavior Under Contexts. In *ICSE*, 2014.

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

[53] M. Zheng, P. P. C. Lee, and J. C. S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *DIMVA*, pages 82–101, 2013.

[54] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, pages 185–196, 2013.

[55] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE S&P*, pages 95–109, 2012.