Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique

Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang

***Abstract*—Although a previous study shows that existing Anti-malware tools (AMTs) may have high detection rate, the report is based on existing malware and thus it does not imply that AMTs can effectively deal with future malware. It is desirable to have an alternative way of auditing AMTs. In our previous work, we use malware samples from Android malware collection GENOME to summarize a malware meta-model for modularizing the common attack behaviors and evasion techniques in reusable features. We then combine different features with an evolutionary algorithm, in which way we evolve malware for variants. Previous results have shown that the existing AMTs only exhibit detection rate of 20%-30% for 10,000 evolved malware variants. In this paper, based on the modularized attack features, we apply the dynamic code generation and loading techniques to produce malware so that we can audit the AMTs at runtime. We implement our approach, named MYSTIQUE-S, as a service-oriented malware generation system. MYSTIQUE-S automatically selects attack features under various user scenarios and delivers the corresponding malicious payloads at runtime. Relying on dynamic code binding (via service) and loading (via reflection) techniques, MYSTIQUE-S enables dynamic execution of payloads on user devices at runtime. Experimental results on real-world devices show that existing AMTs are incapable of detecting most of our generated malware. Last, we propose the enhancements for existing AMTs.**

***Index Terms*—Android feature model, defense capability, malware generation, dynamic loading, linear programming**

I. INTRODUCTION

ACCORDING to a report from AV-TEST [1], the independent IT-security lab, 26 off-the-shelf anti-malware tools (AMTs) show high detection rate (DR) of above 90% for existing Android malware. This test report proves that the mainstream signature-based ATMs can effectively detect *existing* malware, provided with a comprehensive list of malware signatures. However, generally, the development of AMTs usually lags behind the advance of new attack or malware variants. The consequence of the arms race in Android security leads to the sophisticated malware, which may contain a variety of attack behaviors and evasion techniques (e.g., multiple-level obfuscation [2, 3], new transformation attacks [2, 3] and collusion attacks [4, 5]). Besides, dynamically loaded malware is becoming increasingly severe. Existing benchmarks GENOME [6] and DREBIN [7] are not updated to the aforementioned attack or evasion features.

Y. Xue, G. Meng, Y. Liu, J. Zhang and H. Chen are from Nanyang Technological University, Singapore, 639798 (e-mail: {tslxuey,gzmeng,yangliu,zhangj}@ntu.edu.sg and hchen017@e.ntu.edu.sg, see https://sites.google.com/site/malwareasaservice/contact).

T.H. Tan is from Acronis Software, Singapore, 038988 (e-mail: tianhuat.tan@acronis.com).

J. Sun is from Singapore University of Technology and Design, Singapore, 487372 (e-mail: sunjun@sutd.edu.sg).

Several existing studies relate to Android malware generation. DROIDCHAMELEON [2, 3] integrated three types of transformation techniques to generate obfuscated malware, which were used to audit the AMTs. ADAM [8] adopted repackaging and obfuscation techniques to generate different variants for a malware sample. Besides new evasion techniques, mutation is also a common approach to generate new malware. Aydogan and Sen [9] proposed to generate Android malware with a genetic algorithm. The newly generated malware came from the crossover and mutation of malware in GENOME [6], and they conducted experiments to show that the new malware variants can easily bypass the detection of AMTs. Cani *et. al.* [10] used μGP to automatically create new malware undetectable for AMTs, and injected malicious code into benignware to create a Trojan horse.

To sum up, the aforementioned studies mainly adopt new evasion techniques or mutate malware samples for new possible variants. As shown in the study [10], using genetic programming (GP) to mutate malware faces one critical problem: deciding whether an evolved variant still retains the characteristics of malware is a major issue of the evaluator. Behavioral modification of existing malware via GP can neither guarantee the maliciousness of the generated one, nor produce malware with the desirable attack behaviors in a systematic way.

A desirable malware benchmark for AMT auditing should label each sample with the contained fine-grained *attack features*. We refer to *attack feature* (AF) as a step or a component (i.e., triggers, permissions or concrete behaviors) of a certain attack, which links to the configuration or implementation of the functional requirements (intention) of malware. For example, phishing malware usually contains three AFs: a faked GUI that tricks users to input the credentials, a source component to steal the credentials, and a sink component to leak the credential. Neither GENOME [6] nor DREBIN [7] explicitly labels the AFs inside each malware sample, not to mention allowing security analysts to derive new malware variants for auditing AMTs.

In our previous paper [11], Android malware generation is treated as a software product line engineering (SPLE) problem [12], considering new malware variants as product variants in software product line (SPL). We separate each common attack behavior into a basic reusable feature via domain analysis [13] — to modularize the AFs of malware (§ III). In this way, we develop a meta-model (i.e., feature model in SPL, see § II-A and Fig. 2) of Android malware by modularizing AFs into *building blocks*. With SPL for malware generation, having a large set of valid and well-labeled malware is not challenging.

Our previous study shows that existing AMTs are susceptible

to *new variants* of old GENOME malware [11]. The AMTs can detect 90% of GENOME malware on average. After we apply multi-objective evolutionary algorithm (MOEA) to combine different attack and evasion features that are modularized from GENOME, the DR sharply drops to 20%-40% in 10 generations of evolution. Finally, for malware variants after 100 generations, existing AMTs only detect 10%-20% of them on average [11].

However, our tool MYSTIQUE used in previous study does not produce the attack of dynamically loaded malware [11]. Considering the severity of this attack [14], we want to audit whether the AMTs can detect the evolved malware that is assembled and loaded dynamically. Hence, in this study, we extend MYSTIQUE to be service-oriented and name it as MYSTIQUE-S. It adopts dynamic software product line (DSPL) techniques [15] and delivers the generated malware at runtime from the remote server to the client for evading detection.

Technically, MYSTIQUE-S consists of three major steps. First, its client app collects some hardware and software information on device, which is achieved by a simple scanning without root privilege. Then the information is sent to the server side of MYSTIQUE-S (§ IV). Next, the server automatically selects a set of AFs that satisfy the constraints on the user device, and generates the malicious code on the fly (§ V). For example, the details of the user scenario (e.g., the model of device, OS version and installed AMTs) are analyzed and converted to constraints. To guide the AF selection, we propose three goals: *aggressiveness*, *latency*, and *detectability* (§ V-B). Each AF has a score for latency and a score for detectability. Linear programming (LP) is applied to find the AFs that satisfy the constraints and optimize the three goals. Lastly, the malicious code is delivered to the client device via a web service, and executed via the reflection mechanism (§ VI). We adopt the reflection mechanism offered by DEXCLASSLOADER [14], which can load *dex* files and execute the *class* files inside.

Different from the work in [16] which focuses on requirements in malware ontology analysis, we maintain the traceability between the AFs and their corresponding code. To assemble the code of different features, we introduce the behavior description language (BDL) (§ VI-A and VI-B) to serve as the bridge between the high level AFs and the low level implementation code. Owing to the BDL, we validate and generate malicious code in a model-driven way. Compared with previous studies on auditing AMTs using different evasion or obfuscation techniques [2, 3, 8] or at certain time point [17], our studies aim to investigate the impact of various AFs and evasion features (e.g., dynamic loading technique) separately.

Beyond our previous work [11], we also make the following novel contributions in this study:

- In previous study, only for the attack of privacy leakage, we build the malware meta-model and generate the variants [11]. Now, we complement the meta-model with more attacks such as financial charge, phishing and extortion. We modularize the AFs of these attacks, and generate variants accordingly.
- MYSTIQUE-S adopts a service-oriented architecture to collect the client-end data and deliver the malware at runtime. Meanwhile, to support the model-driven malicious code generation, we propose the BDL to glue the high level features with their low level implementation code.
- Our work in [11] relies on MOEA, which is computationally costly. In this work, we adopt linear programming (LP) to select suitable AFs for optimizing the objectives of malware inventor, since LP can rapidly solve the constraints of feature model on the fly and avoid the evolution time of MOEA.
- Instead of using static detection or dynamic detection via virtual machine in the report [11], we evaluate our tool on 16 real Android devices. We observe that in most cases, the malicious code generated by MYSTIQUE-S are not detected. According to our findings, we propose some enhancements for the AMTs.

II. BACKGROUND

A. Dynamic Software Product Line

SPLE is a software development paradigm that has received much attention in the last decade [13]. SPLE aims to reuse the commonality among the products inside the same family, and maintain the product variants in a systematic way. SPLE usually adopts the feature-oriented domain analysis (FODA) to identify the codebase and variant features [18]. The *codebase* refers to the same code shared by all the product variants, which is the implementation of the basic functionality of a software family (a set of similar products) [12]. *Variant features*, which are different extra functions, are used to satisfy the needs of various customers. Typically, SPLE includes two stages: *domain engineering* that builds the architecture consisting of the codebase and variant features, and *application engineering* that derives new products by applying variant features onto the codebase. Generally, automation of product derivation is the main advantage of SPLE.

Feature model (FM). The centric concept in SPLE is to extract the *feature model* [18] — a tree-like feature hierarchy that captures the structural and semantic relationships between features inside. The core task in application engineering is to choose a set of features to derive valid products that satisfy all the constraints [12] and optimize the product performance.

Given a feature f and its sub-features $\{f'_1, ..., f'_n\}$, there exist four types of tree-structure constraints (TCs) (see Fig. 2 for example). We list them and show their logical formula [19]:

- f'_i is a *mandatory* sub-feature — $f'_i \Leftrightarrow f$,
- f'_i is an *optional* sub-feature — $f'_i \Rightarrow f$,
- $\{f'_1, ..., f'_n\}$ is an *or* sub-feature group — $f'_1 \vee f'_2 \vee ... \vee f'_n \Leftrightarrow f$,
- $\{f'_1, ..., f'_n\}$ is an *alternative* sub-feature group — $(f'_1 \vee f'_2 \vee ... \vee f'_n \Leftrightarrow f) \wedge \bigwedge_{1 \leqslant i < j \leqslant n} (\neg(f'_i \wedge f'_j))$.

Further, given two features f_1 and f_2, three types of cross-tree constraints (CTCs) exist, i.e., *requires*, *excludes* and *iff* [19]:

- f_1 *requires* f_2 — $f_1 \Rightarrow f_2$,
- f_1 *excludes* f_2 — $\neg(f_1 \wedge f_2)$,
- f_1 *iff* f_2 — $f_1 \Leftrightarrow f_2$.

In traditional SPLs, variant features are bound to different products statically at compilation time (before the execution of the system). In contrast, adaptive systems support feature binding at runtime and are called dynamic SPLs (DSPLs) [15]. A recent progress in SPLE is the implementation of DSPL via the rapidly emerging paradigm of service-orientation (SO). By

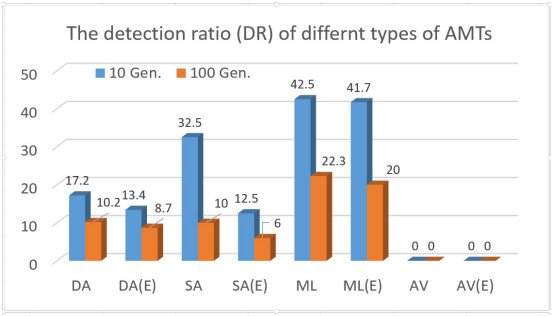


Fig. 1: Results of AMTs auditing by using MYSTIQUE [11]

virtue of the dynamic composition of service, variants features can be loaded into the system dynamically according to user preferences and environmental scenarios. In SPLE, a feature model (e.g., that of Linux kernel) may contain thousands of features. It is a non-trivial problem to select an optimal set of features which satisfies the constraints (i.e., TCs and CTCs) among features. Selecting an optimal feature set represents a searching problem [20]. Such problem is normally addressed in SPLE community using techniques such as MOEAs.

B. Android Attacks

We have witnessed the rapid development and evolution of Android malware since the first Trojan malware was discovered in 2010 [21, 22]. In this paper, we present four types of attacks which are prevailing in the last two years. According to [23], these four types of attacks constitute 60% of Android attacks.

Privacy leakage. Android malware may steal sensitive information on Android devices, such as SMS messages, contact information, geography locations and call logs [24]. The stolen information can be used to track users, make profits, obtain Mobile Transaction Authentication Number (mTAN) and so on. Privacy leakage constitutes a large portion of Android malware (about 78.7% in GENOME).

Financial charge. Premium Rate Services (PRS) are value-added services provided by a telecom provider. PRS include subscriptions to information, services of gaming, charity donations and so on, which charge users beyond the standard network charges. Android malware can stealthily text or call a premium number, and cause extra fees [25].

Phishing. This attack uses social engineering techniques and disguises malware to be a normal app, which tricks users into exposing their credentials. Phishing is becoming progressively severe, after it targets the financial apps [26]. SmiShing, a kind of phishing attacks, spreads fake SMS to users and tricks them into opening the crafted phishing web page and entering their credentials. In addition, malware can also mimic GUIs of target apps (e.g., banking apps and social apps). The credentials entered by users in the fake app will be sent to the attacker.

Extortion. Since ransomware Simplocker was firstly discovered in 2014 [27], plenty of variants have swarmed into mobile devices. Extortion attack in ransomware basically contains two steps — *encrypting* the files in the accessible storage via cipher; *deleting* the original files. After receiving the ransom from the user, the attacker may (or may not) release the encryption key for the victims to decrypt the files.

It is observed that malware variants often share similar code, especially for variants of the same attack. As reported by Crussell *et al.* [28], software clones are common. Recently, Chen *et al.* [29] detect malware based on the clone detection techniques. Thus, code clone analysis helps to identify common malicious code among variants [30]. In [11], relying on code clone analysis on malware variants of privacy leakage [30], we adopt FODA to modularize attack behaviors (and their code) into AFs. In this work, we conduct the same analysis for malware of the three other attacks.

C. Summary of Previous Study

In the previous study, we apply MOEA to mimic malware evolution [11]. In particular, two genetic operators are applied on the current generation to produce next malware generation: *gene crossover* (i.e., exchanging (attack or evasion) features of two samples) and *mutations* (i.e., mutating the selection of features of malware). To retain evasiveness and aggressiveness of malware in evolution, we define multiple evolution objectives (a.k.a. fitness functions) for selecting malware variants to survive into the next generation: 1) maximizing the number of attack behaviors, 2) minimizing evasion techniques needed and 3) minimizing the expected detection rate.

In Fig. 1, we summarize the results by two bars of each of four types of AMTs. The first one is the DR for evolved malware without evasion features; the second one "(E)" is the DR for malware with evasion. "DA" denotes for dynamic based AMTs; "SA" for static based AMTs; "ML" for machine learning based AMTs; "AV" for the popular Anti-virus tools. After malware evolves from 10-*th* to 100-*th* generation, the DR of the audited AMTs sharply dropped. We attribute the low DR for evolved malware to the modularity offered by MYSTIQUE.

In this study, we extend our work in [11] to support more types of attacks and the dynamic loading technique for advanced evasion [14]. To improve the efficiency, in MYSTIQUE-S, we adopt LP (not MOEA) to select AFs for malware generation.

III. FEATURE MODEL OF ANDROID MALWARE

To create new malware variants by reusing the attacks in existing malware, we first analyze the malicious code in malware benchmark GENOME [6] and recent malware samples. Then, we represent AFs as a feature model (FM) via FODA aided by the domain knowledge of security experts. In general, we categorize the AFs into three types, namely trigger, permission and behavior features in § III-A.

For the four types of Android attacks introduced in § II-B, the corresponding FM is partially shown in Fig. 2 under the *behavior* node. Currently, we identify and modularize 93 AFs (§ III-A), and extract the CTCs among these features. For the completeness of the classification, owing to the extendibility of the FM, we can always add new AFs into the FM (e.g., privilege escalation [31]). Note that the FM is a conceptual modeling of features, and we also keep the traceability between a feature and its modularized code in our built SPL (§ III-C).

A. Attack Features

We identify different AFs according to the context, permission and functionality relevant to the attack, as shown below.

Trigger features refer to the configurations that customize the entry points for malicious attack behaviors. Triggers can be

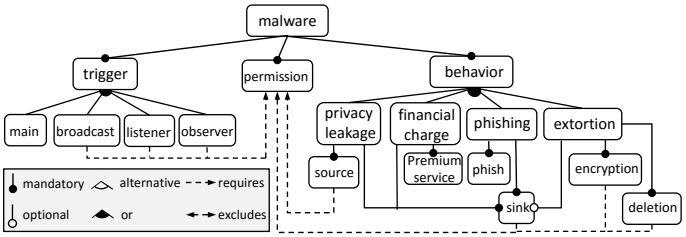


Fig. 2: The partial feature model of Android malware

TABLE I: Parts of attack features in covered attacks

Source	**Category**
TELEPHONY	IMEI, IMSI, PHONE_NUMBER, etc
SMS	INBOX, INCOMING_SMS, etc
CALL	CALL_LOG, INCOMING_CALL, etc
BROWSER	BROWSER_HISTORY, etc
MEDIA	RECORD_AUDIO, etc
LOCATION	REAL_TIME_LOCATION, etc
BUILD	CODE_NAME, SDK, etc
CONTACT	CONTACT, etc
ACCOUNT	ACCOUNT, etc
STORAGE	EXTERNAL, etc
PACKAGE	INSTALLED_APK, etc
Sink	**Category**
HTTP	APACHE_GET, SOCKET_GET, etc
SMS	SEND_TEXT_MESSAGE, etc
Premium service	**Category**
SMS	SEND_TEXT_MESSAGE, etc
CALL	OUTGOING_CALL, etc
Encryption	**Category**
ENCRYPT	ENCRYPT_AES, ENCRYPT_DES, etc

GUI-based or non GUI-based [32]. GUI-based triggers can be easily identified by end users or AMTs, since it requires interaction with visible GUI components [33]. In this paper, we only consider non GUI-based triggers that have no interactions with users. Four types of triggers are identified in GENOME:

- Trigger feature main means that the related behavior can be triggered since the beginning of the life-cycle of an app;
- broadcast means that the behavior is triggered when a broadcast message is received;
- listener means that the behavior is triggered when the registered listener captures a change on the device states;
- observer refers to the observer that is registered on ContentProvider. The observer triggers the behavior when the content provider is changed.

Permission features refer to the permission required for the malware to conduct malicious behaviors [34]. Android provides a permission-based mechanism to avoid the abuse of system sensitive operations. Many malicious behaviors in malware require certain permissions to achieve attack goals. For example, the permission android.permission.READ_PHONE_STATE is required to obtain the IMEI code of the device via invoking the method getDeviceId. In general, information leakage requires the permission for accessing and sending out the information. Similarly, phishing also requires such permissions. Financial charge requires the permission for sending SMS to a number that binds to premium rate services. The recent extortion attack needs to access the public files on device with the permission android.permission.WRITE_EXTERNAL_STORAGE.

Behavior features refer to the malicious behaviors conducted by the attack, to which trigger and permission features are all assistant [35]. Behavior features are the core AFs that mostly link with the modularized malicious code. For the four types of attacks shown in § II-B, there are four types of behavior features, respectively. For each type of behavior features, several steps need to be carried out for the success of the attack. For each attack step, it may have several sub-features that represent different implementations. For instance, in privacy leakage, two steps are carried out: obtaining the privacy (i.e., feature source) and leaking the privacy (i.e., feature sink). For feature source, there are multiple candidate sub-features (SMS information, device information, etc.). Similarly, for feature sink, the leaked privacy can be sent out by SMS or HttpRequest.

Note that the partial FM in Fig. 2 mainly illustrates the high-level organization of these features. Each feature at the bottom level in Fig. 2 may have several sub-features, e.g., feature Source has 11 variant sub-features in an *Or* relationship. Each variant feature in Table I may have several sub-features of different implementations (modularized code) in an *Alternative* relationship. Interested readers can refer to our tool website [36] for the complete FM, the full list of CTCs among the features.

B. Model Extraction

We design the feature model of Android malware based on our manual analysis on the benchmark GENOME, and perform a lightweight static analysis on malware to extract specific features and associated implementation instances.

1) Model Architecture: Inspired by [6], we represent Android malware with three necessary elements as discussed in the previous section. From a high level of perspective, an attack needs to satisfy certain external conditions first, and then be waken up by triggers to execute specific malicious behaviors. The types of triggers are concluded from many previous works [6, 37, 38], and all of them are defined either in the manifest file or in the implementation. For simplicity, we only take into account permissions as configurable conditions that guarantee the execution of malicious behaviors. The types of malicious behaviors comply with the mainstream classification of attacks in the mobile world [6, 23, 26].

2) Feature Extraction: As GENOME is well-known and studied for the malicious code inside the malware, manual analysis of GENOME malware is feasible and effective. Still, some manual effort is needed to derive 16 common attack features (93 variant features at the implementation level). However, after we manually identify the malicious features of some samples, owing to the aid of some tools, we can scale this by the semi-automatic process[1]. We perform a lightweight static analysis on Android malware to extract the concrete implementations (variant features) for each common feature. The three kinds of sub-features introduced in the previous section are extracted as follows:

- permission features, which can be extracted directly from the manifest file. Additionally, we remove out self-defined permissions and focus on dangerous permissions[2];

[1]In the tool website, details are provided on how attack features are derived manually or semi-automatically. To see an example — how attack features are grouped, variant features are introduced, and how composability is handled — interested readers can refer to this link: https://sites.google.com/site/malwareasaservice/home/featuremodel

[2]https://developer.android.com/guide/topics/security/permissions.html#normal-dangerous

- trigger features can be inferred either from the manifest file or the implementation. For example, a broadcast receiver can be defined as "⟨receiver⟩" in the manifest file, or dynamically registered by registerReceiver in the code. Similarly, other triggers can be extracted automatically from malware;
- behavior features, that are extracted from the code. Based on [34] and acquired permissions in malware, we locate the invocation of sensitive APIs, and subsequently identify the usage patterns of these APIs as feature instances. Since dynamic code loading has been already widely used in malware, some invocations of sensitive APIs may bypass our scanning. Therefore, we specifically investigate the reflection employed in the code and interpret the real invocations.

Note that we modularize the common attack features that explicitly own malicious code (in particular, malicious code in Java), and the code of the covered attacks in this paper. For malware GingerMaster that employs native code to gain the root privilege, of which the attack is not considered in the paper, we do not elicit malicious functionality from them.

The current FM is built according to the availability of an attack and its possible implementation instances in GENOME. For example, there are many implementation instances of AF of privacy leakage. There are many sources of sensitive information such as data that can be obtained by invoking Android APIs, data that is stored in Content Provider, and data that is sent by system broadcast of incoming SMS. Similarly, there are many implementation instances for sink features and ways to link the source and the sink. Therefore, we have constructed most attacks of privacy leakage. The attack of financial charge in GENOME basically sends a specific message to a premium rate number. The primary difference is the parameters of either the sent message or the premium rate number, and hence the implementations are quite similar. Thus, we only construct one sample to represent the attack of financial charge. The situation is similar to the attack of phishing. For the attack of extortion, it actually does not exist in GENOME. Considering its emergency and increasing popularity, we construct one sample for extortion. Since there are few samples and variants for analysis nowadays, it is also parameterized for more variants like financial charge. Nevertheless, owing to the extendibility of the FM and Mystique-S, new variant features for an attack (e.g., extortion) can be added when more implementation instances of the attack are available.

C. Feature Modularization

The code of AFs is modularized into code units of various granularity, ranging from several packages to a single method. The phishing AF usually contains the largest number of lines of code (LOC), as it has the faked GUI or functionalities to deceive the users. Hence, the corresponding code of phishing attack can be close to the genuine app, with the LOC up to a reasonably large number. In contrast, the implementation of financial charge (or adware) can be just several lines of code and easily modularized into a method. For example, in Fig. 3, we show the modularized code for sending the token by SMS (*D1*). The token is intercepted by registering a BroadcastReceiver and listening to the incoming SMS messages and then sent out in an SMS message to a specific number via SmsManager.

```
public void onCreate(Context context, Intent intent){
   if (intent.getAction().equals("android.provider.Telephony
        .SMS_RECEIVED")){
      final Bundle bundle = intent.getExtras();
      if (bundle != null) {
         final Object[] pdusObj = (Object[]) bundle.get("
              pdus");
         for (int i = 0; i < pdusObj.length; i++) {
            SmsMessage currentMessage = SmsMessage.
                createFromPdu((byte[]) pdusObj[i]);
            sb.append(currentMessage.getDisplayMessageBody()
                ).append("&");}
      }
   }
   SmsManager sm = SmsManager.getDefault();
   sm.sendTextMessage(number, null, message, null, null);
}
```

Fig. 3: The modularized code of sending token via SMS (*D*1)

IV. RUNNING EXAMPLE AND SYSTEM OVERVIEW

A. A Motivating Example

Fig. 4 depicts an exemplar of malware service that dynamically loads malicious code from a remote server[3]. The basic idea is to disguise the client app as a benign app that tricks users into entering credentials and then intercepts the SMS with two-factor token. The basic steps are executed as follows:

1. After the client app is installed on user device, it starts a daemon service to communicate with the service provider of malware. It collects and sends the user information (e.g., hardware and software information of the device) to the server, and receives the malicious payloads from the server.
2. After the malicious code is delivered to the client, the daemon service starts a fake bank activity from the component *A* inside (**Step 1** in Fig. 4). In the life-cycle of the phishing activity, two code snippets are instrumented into the component *B* and *C*, respectively.
3. The code in *B* is to change the view of activity to mimic the specific bank app, and the code in *C* is to get the entered credentials and send them to the server (**Step 2**).
4. Last, the daemon service registers a broadcast receiver to listen to incoming SMS messages (**Step 3**). The SMS message that contains the two-factor token (the key for two-factor authentication) is leaked to the attacker (**Step 4**).

As shown in Fig. 4, for the same attack step, there may exist various implementations, which are also regarded as candidate AFs. For example, for the phishing attack in Fig. 4, there exist three AF candidates (i.e., different views) of phishing attack (*B1, B2, B3*). For feature *LeakCredential* in component *C*, there are two AF candidates: sending credentials by Apache connection (*C1*) and sending them by SMS (*C2*). For the feature *LeakToken* in component *D*, there are two candidates: sending token via SMS (*D1*) and sending token via socket (*D2*). For simplicity, in this example, we just show two or three candidate AFs for each attack step, and also omit the finer-grained AFs of the source and sink operations at step 2 and 3. Types and granularity of AFs are explained in § III.

For this example, three tree constraints (TCs) and five cross-tree constraints (CTCs) need to be satisfied. For example, TC_2 means if *LeakCredential* is selected, at least one of *C*1 and *C*2 must be selected, and vice versa. CTC_3 means the selection of *LeakToken* requires the selection of permission feature *P*3.

[3]The original version of the example malware is found in March 2016 [39], but it is neither service-oriented nor dynamically loaded.

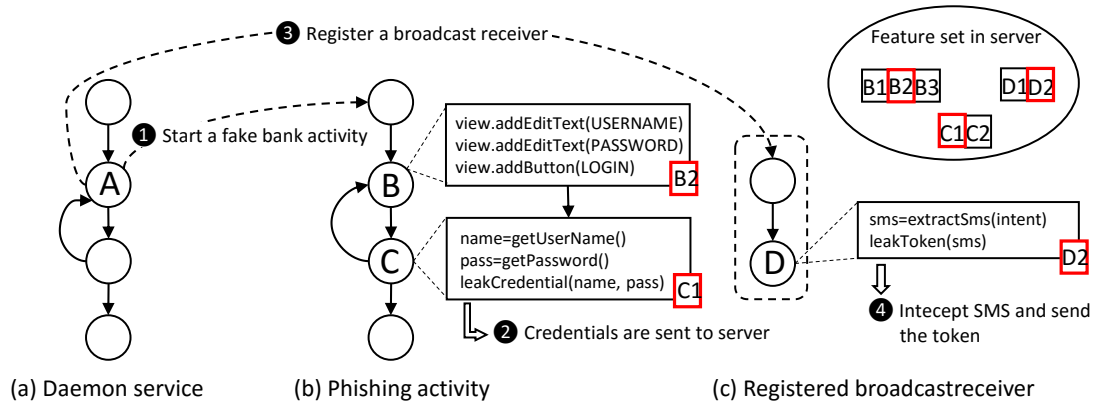


Fig. 4: A running example of MYSTIQUE-S

$TC_1 : B1 \vee B2 \vee B3 \Leftrightarrow Phish$	— *or* type
$TC_2 : C1 \vee C2 \Leftrightarrow LeakCredential$	— *or* type
$TC_3 : D1 \vee D2 \Leftrightarrow LeakToken$	— *or* type
$CTC_1 : C1 \Rightarrow P1$ (*android.permission.INTERNET*)	— *require* type
$CTC_2 : C2 \Rightarrow P2$ (*android.permission.SEND_SMS*)	— *require* type
$CTC_3 : LeakToken \Rightarrow P3$ (*android.permission.RECEIVE_SMS*)	— *require* type
$CTC_4 : D1 \Rightarrow P2$ (*android.permission.SEND_SMS*)	— *require* type
$CTC_5 : D2 \Rightarrow P1$ (*android.permission.INTERNET*)	— *require* type

The suitable AFs need to be automatically selected for the sake of a better success ratio of attack, given different user scenarios (e.g., model of device, OS version and installed AMTs). For example, if the device installs NORTON, AFs {*B2*, *C1*, *D2*, *P1*, *P3*} should not be selected. The reason is that NORTON reports suspicious apps based on the permission *P2*. If no AMT is installed, AFs (*B2*, *C2*, *D1*, *P2*, *P3*) are selected as *P2* can be selected for short latency due to the immediate action of sending SMS messages.

B. System Architecture

MYSTIQUE-S is a framework of automated malware generation, which takes as input the client-end contextual information and outputs the user-tailored malicious code. Based on the Android malware FM (§ III), MYSTIQUE-S automatically selects AFs according to the user scenario via linear programming (LP in § V-C). Then, the selected AFs guide the model-driven generation of malicious code (§ VI). Last, the *payloads* are delivered to the user device and loaded dynamically (§ VI). Here, *payloads* refer to the generated malicious code and the corresponding *instructions* (i.e., the command for the client app to load the code of AFs in sequence).

Fig. 5 depicts the architecture of our tool, which contains three parts as we discuss below:

- **Client app.** Its task is to (periodically) collect the contextual information on the user device, receive malicious code and instructions from the server, and launch the attack by using the dynamic code loading mechanism (§ VI-C). As shown in Fig. 5, three critical modules are included in the client app: 1) *daemon service* interacts with the service provider and starts an attack once receiving the malicious code and instructions. 2) *dynamic instrumentation* deploys the malicious code in different components (e.g., Intent) of the client app, interprets the received instructions and acts accordingly. 3) *execution of malicious behavior* executes the instructions (loading the malicious code of multiple AFs in sequence). Finally, the execution results are fed back to the daemon service.

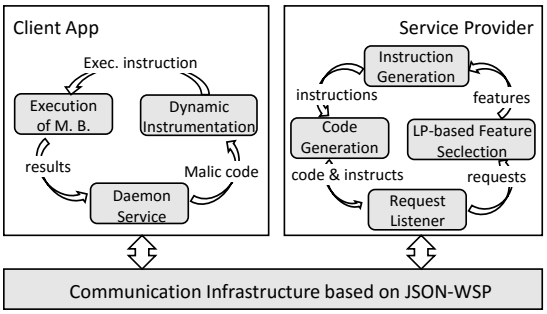


Fig. 5: The overview of system

- **Service provider.** The service provider listens to the requests from the client app on installed devices. After receiving the user device information, it selects AFs and generates the corresponding payloads. Four modules are involved in the process: 1) *request listener* receives attack requests and initializes the automatic generation of payloads. 2) *LP-based feature selection* selects an optimal combination of AFs from the Android malware FM. 3) *instruction generation* takes input as the selected AFs and generates the instructions by considering the context in the client app. One instruction, in the format of BDL that specifies the workflow of malware (§ VI-A), contains the execution context and the operation to execute. 4) *code generation* generates the malicious code by assembling the code of AFs according to the BDL. After the process, *request listener* sends the generated payloads to the daemon service on user device.
- **Communication infrastructure.** It provides a connectionless protocol that enables the asynchronous communication between the client app and the server. As an attack needs multi-round interactions between the client app and the server, the connection is not retained during the lifecycle of an attack for the sake of hiding the attack. Instead, the service provider will track the state where the attack proceeds. In addition, the exchange message follows the standard JSON-WSP [40] for a bidirectional communication (see § VI-C).

V. USER-TAILORED ATTACK FEATURE SELECTION

In this section, we explain how the user-tailored AFs are automatically selected by linear programming (LP). First, we show how to convert TCs and CTCs among features to inequalities for LP based constraint solving (§ V-A). Then we define the malware generation goals (§ V-B). Last, we resolve

the AF selection problem via LP (§ V-C), i.e., satisfying the inequalities and optimizing the objective functions.

A. Converting Features Constraints to Binary Inequalities

To select features and generate the products that satisfy the TCs and CTCs (defined in § II-A) inside the FM, Broek [41] adopted integer programming (IP) for the feature selection problem (i.e., initialization of valid product in [41]). Broek converted the TCs and CTCs into the integer inequalities, and then apply IP to resolve these inequalities. In this paper, we further convert the TCs and CTCs into the inequalities of binary variables. Given a feature f, the binary value represented by the selection of f (denoted as $|f|$) is **1** if selected, and otherwise $|f|$ is **0**. According to the integer inequalities deduced in [41], we can further deduce the corresponding binary inequalities for the TCs and CTCs. In Table II, we list the binary inequalities for different types of constraints. Due to the page limit, we only provide the proof on our website [36].

TABLE II: Binary inequalities for different types of constraints

Constraint Type	Binary Inequality
f and its *mandatory* sub-feature f'	$\|f'\| - \|f\| = 0$
f and its *optional* sub-feature f'	$\|f'\| - \|f\| \leqslant 0$
f and its *or* sub-features	$\forall i \in \{1,...,n\}$ $\|f'_i\| - \|f\| \leqslant 0$ $\sum_{i=1}^{n} \|f'_i\| - \|f\| \geqslant 0$
f and its *alternative* sub-features	$\forall i \in \{1,...,n\}$ $\|f'_i\| - \|f\| \leqslant 0$ $\sum_{i=1}^{n} \|f'_i\| - \|f\| \geqslant 0$ $\sum_{i=1}^{n} \|f'_i\| \leqslant 1$
f_1 *requires* feature f_2	$\|f_1\| - \|f_2\| \leqslant 0$
f_1 *excludes* feature f_2	$\|f_1\| + \|f_2\| \leqslant 1$
f_1 *iff* feature f_2	$\|f_1\| - \|f_2\| = 0$

B. Goals of Attack Feature Selection

Apart from the constraints to satisfy, we also need to define the design goals for malware generation. We propose three objectives to guide the AF selection: *aggressiveness*, *latency*, and *detectability*. As the results of AF selection, malware is getting more aggressive with shorter latency, but being less detectable. Given a solution $\vec{x}$, we represent it as a bit vector of all AFs, where $\{f_1...f_n\}$ denotes the set of n AFs. The objective functions are defined as follows.

Obj1. *Aggressiveness*: to make the malware more aggressive, we want to minimize the number of AFs that are not selected. It is defined as: $\mathcal{F}_1(\vec{x}) = \sum_{i=1}^{n}(1 - |f_i|)$.

Obj2. *Latency:* to shorten the time-delay in attack launching (e.g., leaking by SMS has less latency than leaking by Internet), we aim to minimize the total latency of all selected features. It is defined as: $\mathcal{F}_2(\vec{x}) = \sum_{i=1}^{n}(|f_i| \times l_i)$, where l_i denotes the latency of AF f_i.

Obj3. *Detectability:* to increase the chance for malware to succeed, we minimize the probability to be detected by AMTs. It is defined as: $\mathcal{F}_3(\vec{x}) = \sum_{i=1}^{n}(|f_i| \times d_i)$, where d_i denotes the detection ratio of AF f_i if f_i is applied alone.

Intuitively, *Obj1* and *Obj2* are competing with *Obj3*, meanwhile *Obj1* and *Obj2* are mutually competing. For instance, having more attacks or shorter latency will lead to earlier and easier detection of the attack. Besides, having more AFs, which is desired, can lead to an undesired side effect of higher latency. With the feature constraints in § V-A that are linear, the three objective functions are also linear. Hence, LP can be applied to resolve this optimization problem. Note that $l_i \in [0,3]$ and $d_i \in [0,10]$ are empirical values, according to our preliminary studies. For example, latency l is set to 1 for $C1$, and 2 for $C2$; detectability d is set to 3 for $C1$, and 4 for $C2$. More discussions on the setup of values of l_i and d_i can be found in § VIII.

C. Attack Feature Selection via LP

For the richness of possible solutions, we would not encode three objectives into one weighted objective for one time solving. Instead, we treat each objective equally and solve this *Multi-objective Optimization Problem (MOP)* using the Pareto dominance relation [42]. In MOPs, usually there exists no single solution that simultaneously optimizes all objectives. Hence, we are interested to find the *non-dominated solutions*. A solution is called non-dominated, if none of the objectives can be improved in value without degrading other objectives [42].

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

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

Subject to the inequalities on variables ($|f_1|...|f_n|$ in our case), where $\vec{\mathcal{F}}$ is a k-dimensional objective vector, $\mathcal{F}_i(\vec{x})$ is the value of $\vec{\mathcal{F}}$ for i-th objective, and $\vec{x}$ is the feature set $\{f_1, ..., f_n\}$.

Technical innovation. To resolve MOPs, MOEAs are often applied [43, 44]. MOEAs are generally scalable, but it requires some evolution time. As heuristic search techniques, MOEAs cannot guarantee to find many non-dominated solutions. Traditionally, LP can only solve single-objective LP optimization. Considering the manageable feature size of the FM (§ III), we apply LP to resolve the MOPs in an analytic way.

The basic idea is that: we retain an objective as the goal function for optimization, and convert two other objective functions into constraints by setting the concrete bounds for them. To find more non-dominated solutions, we need to gradually adjust the bounds for these two objective functions.

Algorithm 1 depicts the main process of LP-based AF selection. At lines 1-3, the user information (e.g., the model of device, OS version and installed software) is analyzed via function *getInfor*() and the searching bound for *obj2* and *obj3* are suggested. For the example in § IV-A, if the user device installs many AMTs and the latest Android version, the malware should have a low detection ratio (a small ratio of the theoretic upper bound, e.g., $10\% \times \sum_{i=1}^{n} d_i$), and can tolerate a little high latency (a large ratio of the theoretic upper bound, e.g., $50\% \times \sum_{i=1}^{n} l_i$). At lines 4-9, we gradually adjust the upper bounds of *obj2* and *obj3* and get the corresponding solutions. At line 8, $bintprog(allConsts, obj1)$ is the LP solving function that optimizes $obj1$, subject to the constraints of inequalities in *allConsts*. Finally, it reaches the termination condition (i.e., the upper bounds) and gets the candidate solutions into *solutions*.

For the constraints of the example in § IV-A, according to Table II, we can convert these logical formula to the inequalities for LP solving function *bintprog*. For the termination case of our example, lines 6-9 get the following inequalities and perform

Algorithm 1: Linear programming guided feature selection

Input: *featureMdl*: the feature model of Android malware
Input: *userInfor*: the contextual information of user device
Output: *solutions*: a non-dominated solution set for feature selection
Output: *returnedSol*: a solution returned to guide malware generation

1 $solutions \leftarrow \varnothing$;
2 $upper\_o2 = getInfor(userInfor), lower\_o2 = 0$;
3 $upper\_o3 = getInfor(userInfor), lower\_o3 = 0$;
4 **for** $i = lower\_o2;\ i \leqslant upper\_o2;\ i = i{+}1$ **do**
5 **for** $j = lower\_o3;\ j \leqslant upper\_o3;\ j = j{+}1$ **do**
6 $const\_o2 = convert(obj2, i), const\_o3 = convert(obj3, j)$;
7 $allConsts = TCs \cup CTCs \cup const\_o2 \cup const\_o3$;
8 $nondominatedSol = bintprog(allConsts, obj1)$;
9 $solutions = solutions \cup nondominatedSol$;
10 $returnedSol = solutions.First()$;
11 **for** $sol \in nondominatedSol$ **do**
12 **if** $aggregatedObj(sol) < aggregatedObj(returnedSol)$ **then**
13 $returnedSol = sol$;
14 **return** *returnedSol*;

the LP solving. Note that $|B1|$ returns 1, if $B1$ is selected; $O2C$ is the constraint converted from *obj2*, where $\sum_{i=1}^{n} l_i$ refers to the sum of latency of each feature; $O3C$ is converted from *obj3*, where $\sum_{i=1}^{n} d_i$ refers to the sum of the chance of each feature to be detected.

Minimize $\vec{\mathcal{F}} = (\mathcal{F}_1(\vec{x}))$, where $\vec{x}$ is the set of all features
Subject to: $TC_1 \wedge TC_2 \wedge TC_3 \wedge CTC_1 \wedge \ldots \wedge CTC_5 \wedge O2C \wedge O3C$
$TC_1 : |B1| \leqslant Phish, |B2| \leqslant Phish, |B3| \leqslant Phish, |B1|+|B2|+|B3| \geqslant Phish$
$TC_2 : |C1| \leqslant LeakCredential, |C2| \leqslant LeakCredential, |C1|+|C2| \geqslant LeakCredential$
$TC_3 : |D1| \leqslant LeakToken, |D2| \leqslant LeakToken, |D1|+|D2| \geqslant LeakToken$
$CTC_1 : |C1| \leqslant |P1|$ $\quad$ $CTC_2 : |C2| \leqslant |P2|$
$CTC_3 : |LeakToken| \leqslant |P3|$ $\quad$ $CTC_4 : |D1| \leqslant |P2|$
$CTC_5 : |D2| \leqslant |P1|$
$O2C : 0 \leqslant \mathcal{F}_2(\vec{x}) \leqslant 50\% \times \sum_{i=1}^{n} l_i$
$O3C : 0 \leqslant \mathcal{F}_3(\vec{x}) \leqslant 10\% \times \sum_{i=1}^{n} d_i$

At lines 11-13, among the candidate solutions, we combine several objectives into an aggregated one, by normalizing the ranges of objectives and assigning them with different weights via function *aggregatedObj*() at line 12. At lines 12-13, we iterate all candidate solutions and identify the optimal solution according to the weighting scheme. In addition, in practice, we refine the returned optimal solution by applying some extra constraints, which are not from the FM, but from the observations on AMTs and AFs. For example, if NORTON is installed on the device, feature $P2$ android.permission.SEND_SMS should not be selected — NORTON reports the third-party app as suspicious if it requires $P2$. In other scenarios, if no AMT is installed on the device, AFs (*B2*, *C2*, *D1*, *P2*, *P3*) are selected as $P2$ can be selected for the short latency of sending SMS immediately.

We clarify that to utilize the user contextual information, the relaxed LP approach is proposed to run LP solving for multiple times. With more candidate solutions, the variety of selected AFs (and the generated code) is improved, preventing the signature- or clone-based detection. Instead, directly combining 3 objectives into an aggregated one and solving it once just yields one solution, which impairs the variety and the unpredictability of the selected AFs.

VI. DYNAMIC GENERATION AND EXECUTION OF MALICIOUS CODE

After the server conducts AFs selection via LP, we show how to assemble the corresponding code of AFs via a model-driven way (§ VI-A and § VI-B). Then, we explain how the generated malicious code is sent to the client app via JSON-WSP. Last, it is dynamically loaded and executed at the client end (§ VI-C).

A. Behavior Description Language

Semantics of the selected AFs is represented in a modeling language, named Behavior Description Language (BDL). The BDL representation for the AFs is more implementation oriented. BDL is used for two purposes: on the server side, it bridges the gap between the malware FM and the workable implementations; on the client side, it assures that behaviors of AFs are executed as designed.

Backus Naur Form of BDL. We present the partial BNF of BDL in Fig. 6 (refer to [36] for the complete definition of BDL). An attack can be divided into several subsequential operations, i.e., $\langle ATTACK \rangle ::= \langle FUNCTION \rangle (' \rightarrow ' \langle FUNCTION \rangle)^*$. Hereby, $\langle FUNCTION \rangle$ is the basic step (building block) for an attack, and it denotes the operation to execute as well as the execution context. One function consists of three elements — $\langle COMPONENT \rangle$, $\langle POINTCUT \rangle$ and $\langle OPERATION \rangle$, where $\langle COMPONENT \rangle$ denotes the component, the building blocks of Android apps, $\langle POINTCUT \rangle$ denotes the methods where malicious behaviors are located, and $\langle OPERATION \rangle$ denotes the operation of malicious behaviors. The component and method together identify the execution context for this operation.

Connection between feature and BDL. As the direct assembly of code of the selected AFs may not yield a workable (no compilation or runtime error) malicious code. Hence, BDL is required to bridge the gap between the selected AFs and the code implementation by adding the execution context of AFs and auxiliary behavioral operations in implementation.

Conceptually, among the selected AFs, each behavior feature relates to one $\langle FUNCTION \rangle$ in BDL. As behavior feature is defined at the atomic behavior level (one step of the attack), its corresponding code is usually modularized into the code unit of method. The modularized code of feature conceptually links to one $\langle OPERATION \rangle$. Hence, assembling modularized code of features essentially requires to describe an $\langle OPERATION \rangle$ with the proper $\langle COMPONENT \rangle$ and $\langle POINTCUT \rangle$. For example, one attack of privacy leakage is to steal users' SMS messages. According to the FM, it needs a $\langle FUNCTION \rangle$ to get SMS messages (i.e., source), and a $\langle FUNCTION \rangle$ to send them out (i.e., sink). These two steps comprise this attack. The code method of source is an $\langle OPERATION \rangle$, and this method is invoked in some $\langle COMPONENT \rangle$. The source operation also needs a permission feature android.permission.READ_SMS, and the behavior need to be started in some $\langle POINTCUT \rangle$ — e.g., from bootup of an app (i.e., trigger feature main) or from a change event of a Content Provider (i.e., trigger feature observer).

Hence, BDL can provide details on: the component of activity or service, the method where the malicious code is injected and executed; the data flow from source to sink, using Android lifecycle and Inter-Component Communication (ICC).

⟨*ATTACK*⟩ ::= ⟨*FUNCTION*⟩(′→′ ⟨*FUNCTION*⟩)*
⟨*FUNCTION*⟩ ::= ⟨*COMPONENT*⟩′ ::′ ⟨*POINTCUT*⟩′ ::′ ⟨*OPERATION*⟩
⟨*COMPONENT*⟩ ::= ′ *ACTIVITY*′ |′ *SERVICE*′ |′ *BROADCAST_RECEIVER*′...
⟨*POINTCUT*⟩ ::= ′ *POINTCUT_ONCREATE*′ |′ *POINTCUT_ONSTART*′...
⟨*OPERATION*⟩ ::= ⟨*SOURCE_SIG*⟩ | ⟨*ENCRYPT_SIG*⟩ | ⟨*PHISH_SIG*⟩...

Fig. 6: Parts of BNF for BDL

```
1  class Task{
2   /* Feature declarations */
3   // code of phishing feature B2
4   void phishing(){ ... }
5   // "Sink" of feature C1, send credentials by Apache conn.
6   String sendCredential(String data){... }
7   // "Source" code of feature D2, read incoming SMS.
8   String getIncomingSms(){ ... }
9   // "Sink" code of feature D2, send token by Socket conn.
10  String sendToken(String data){...}
11
12  /* The invocation to features */
13  Object operateOn(String comp, String met){
14    if (comp=="ACTIVITY"&&met=="ONCREATE") {
15      phishing();
16    }else if (comp=="BROADCAST_RECEIVER"&&met=="ONRECEIVE"){
17      sendCredential(getIncomingSms()) ;
18    }
19   ...}
20 }
```

Fig. 7: Generated code for the selected AFs (*B2*, *C1* and *D2*)

B. Model Driven Malicious Code Generation

In MYSTIQUE-S, we have set some rules for automated generation of BDL for selected AFs, including various commonly-used source-sink patterns [38], and information flows for phishing attack. The service provider further interprets BDL to generate the corresponding malicious code. As the malicious code is dynamically loaded and executed in the client app, MYSTIQUE-S will not bind or invoke the code snippets of AFs at server side. Hence, the generated malicious code includes two parts: the declaration of code for AFs (in the format of Java method), and the invocation method to AFs.

An illustrative example. For the example in Fig. 4, it is a composite attack with privacy leakage and phishing. As the phishing feature can only be deployed in the main thread of an activity, it is assigned to the context of ACTIVITY::ONCREATE. The acquisition of incoming SMS messages needs to be done in the context of a registered broadcast receiver. Thus, the selected AFs (i.e., *B2*, *C1*, *D2*) in § IV-A have the corresponding BDL:

ACTIVITY::*ONCREATE*::*PHISH*()
→*ACTIVITY*::*ONCREATE*::*SINK*(*HTTP* :: *APACHE_POST*, *CREDENTIALS*)
→*BROADCAST_RECEIVER*::*ONRECEIVE*::*SOURCE*(*SMS*::*INCOMING_SMS*)
→*BROADCAST_RECEIVE*::*ONRECEIVE*::*SINK*(*HTTP*::*SOCKET_POST*, *LOCAL_VARIABLE*)

Based on the above BDL, MYSTIQUE-S generates the malicious code in Fig. 7. Lines 3-14 provide the declarations for these features, and lines 15-22 present the invocation to these declarations. In method "operateOn", it defines the statements (i.e., the invocations to specific feature declarations) as the instruction of attack for different steps.

C. Dynamic Loading and Execution of Malicious Code

Malicious code is dynamically loaded and executed in the client app. The process relies on two mechanisms as below.

Single-step loading via JSON-WSP. JavaScript Object Notation Web-Service Protocol (JSON-WSP) [40] is a web-service protocol that uses JSON for service description. We use JSON-WSP to exchange messages between client app and the server.

Initially, the service provider generates a sequence of instructions to execute an attack. The client app queries and receives from the server an instruction each time, named *single-step loading*. The main part of instructions contains the type of instructions and the content of the instructions, in the format of {"command":"", "value":""}. There are two types of instructions — *download* that indicates the address of the payload to download, and *execute* that provides a serial of operations in BDL. For the running example, the first instruction received by single-step loading is a *download* instruction to download the malicious code, the following *execute* instruction is to execute the behaviors defined in the BDL. (§ VI-B).

```
1 DexClassLoader loader = new DexClassLoader("[DEX_FILE]", "[
      CACHE_FILE]", "[LIB_PATH]", "[CLASS_LOADER]");
2 Class clz = loader.loadClass("Task");
3 Object obj = clz.newInstance();
4 Method mtd = clz.getDeclaredMethod("operateOn", "[COMP]", "[
      POINTCUT]");
5 mtd.invoke();
```

Fig. 8: A simple example of using reflection mechanism

Dynamic execution via reflection. MYSTIQUE-S employs Java Reflection to dynamically execute the malicious code. Similar with the idea of XPOSED [45], MYSTIQUE-S injects a small code snippet (shown in Fig. 8) into each execution context of Android app. The code then checks the payloads whether there is a task to execute in this current context. As the payloads (e.g., operateOn in Fig. 7) define the operations to do in different contexts, the malicious behaviors are dynamically loaded into a specific context. In Android, reflection is based on the class DEXCLASSLOADER which can load *dex* files and read the included class files. As shown in Fig. 8, the client app needs to create an instance of DexClassLoader by specifying the location of the *dex* file. The class loader is used to instantiate the target class and thereby the target method.

VII. EVALUATION

MYSTIQUE-S is implemented in about 4,187 lines of Java code (23.9% for the client app, 76.1% for the service provider, and modularized AF code is not included). It adopts CPLEX [46] for solving LP. Considering the dynamic attack, experiments are conducted on dynamic Analysis Tools (DATs) or real devices installed with AMTs; the service provider is deployed on a workstation running on Ubuntu 14.04 with Intel Xeon(R) CPU E5-2697 and 64G memory. We aim to answer the following research questions.

RQ1. Are the modularized AFs valid? Is the dynamically assembly malicious code workable at runtime?

RQ2. Can the mainstream AMTs and online vetting process detect the malware dynamically generated by our tool?

RQ3. Is MYSTIQUE-S adaptive to the different attacks in real cases and helpful for the recurrence of an attack?

Evaluation subjects. To evaluate the evasiveness of the dynamic attack and audit the AMTs, we select several state-of-the-art AMTs for detection in Table III and IV.

A. RQ1: Validity of Generated Malicious Code

In this section, we evaluate the validity of MYSTIQUE-S. Specifically, we conduct experiments to show the validity

TABLE III: The detection results of ODTs, where ✓ means "passed" and ✗ means "detected"

Tool	DS#A	DS#B	Tool	DS#A	DS#B
FLOWDROID	✓	✗	ICCTA	✓	✗
DROIDSAFE	✓	✗	NORTON	✓	✓
AVG	✓	✓	AVAST	✓	✓
BITDEFENDER	✓	✓	ESET	✓	✓
KASPERSKY	✓	✓			

of malicious code that is generated from each AF. Further, we evaluate the service-oriented communication mechanism between the server and the client app.

Among the 93 AFs introduced in § III-A, we identify 44 behavior features. For each behavior feature, we select its required permission features and trigger features, and generate the BDL representation. MYSTIQUE-S generates the corresponding malicious code according to the BDL. Then we repackage the malicious code into a blank Android app to wrap it as malware. Finally, we execute the malware on the emulator to verify whether the carried malicious code can be successfully executed. The results show that malicious code can fulfill its malicious intent, e.g., leaking information, extortion. In this experiment, we confirm that each behavior feature, as single building block, is valid and workable on its own.

To confirm the validity of the generated malicious code, a *honeypot* is set up to receive the report of a successful attack (e.g., the stolen information is sent to the honeypot) in the experiment. Our honeypot has successfully received the response from emulators or experimental devices. It proves that our generated malicious code works in practice, which encourages us to conduct user studies on real devices (§ VII-B).

During the communication between the client app and the service provider, multiple sequential instructions are exchanged to complete an attack. The bidirectional communication is asynchronous, which means that the client app may receive and execute only one individual instruction each time. To guarantee the client app has obtained all necessary malicious code and instructions, MYSTIQUE-S employs *periodical querying* in the client app and *state retaining* in the service provider. The daemon service in the client app will periodically enquire service provider to check: 1) it is alive; 2) what to do in the next step. This mechanism avoids the tense work (e.g., high network traffic and high memory usage rate) with launching an attack, and thereby reduces the probability of being perceived by users. After identifying the attack to launch with LP, the service provider retains the state where the attack proceeds. In our experiments, we set the time interval as 30 minutes for periodical querying. Results show that this mechanism can tolerate the loss of Internet connection, and restore the attack state after the client app is reconnected to the Internet.

B. RQ2: Auditing the AMTs on Real Devices

We have evaluated the resistance of generated malicious code to the detection in three aspects: offline detection tools, dynamic analysis tools and AMTs installed on Android devices.

1) Resistance to Offline Detection Tools (ODTs): To evaluate the evasiveness of the client app against ODTs, we choose several state-of-the-art static analysis tools and AMTs from VIRUSTOTAL. To evaluate the efficacy of this dynamic and optimal selection of AFs, we conduct an experiment that uses the client app with/without the payloads, respectively. As shown in Table III, column *DS#A* shows the results of scanning the client app without payloads; column *DS#B* shows results of scanning the client app with payloads. Here, payloads are the malicious code generated according to the 44 behavior features.

Based on our observations from Table III, it is concluded that MYSTIQUE-S can effectively bypass the detection of ODTs. Generally, static analysis collects the evidences in the *apk* file for detection. However, MYSTIQUE-S only dynamically loads malicious code in an attack, and it does not store any malicious code in the *apk* file. Hence, it has a very low probability of being detected by ODTs.

2) Resistance to Dynamic Analysis Tools (DATs): We deploy three state-of-the-art DATs to evaluate the evasiveness of MYSTIQUE-S. These three tools are listed below:

- **DROIDBOX**[4] automatically intercepts and modifies API calls made by a targeted app. It captures the behaviors of apps at runtime, e.g., information leakage, cryptographic operations, the invocations of Android APIs and etc.
- **DROZER**[5] allows to search for security vulnerabilities in apps and devices by assuming the role of an app and interacting with the Dalvik VM.
- **TAINTDROID** [47] can track how apps use sensitive information via *taint analysis*. It has hooked several transfer channels, including memory, file system, and event dispatch.

We construct 22 attacks (requesting specific permissions) of privacy leakage with regard to the types of sensitive information, 1 attack of premium service, 3 attacks of phishing, and 1 attack of extortion. DROIDBOX can successfully capture many behavior logs of the client app, for example, the download of malicious payload, the acquisition of contact and SMS, the operation to send SMS messages (perhaps to a premium rate number) and the cryptographic operation. However, it still needs manual efforts to confirm whether these behaviors are malicious or not. In comparison, DROZER can only identify the started Android components and the acquired permissions of the client app. Since TAINTDROID only targets privacy leakage of apps, it only detects 10 attacks (45.5%) of privacy leakage in our experiment, while it fails to detect other kinds of attacks.

Summary. Compared to static analysis, the DATs can effectively detect attacks via dynamically loaded malicious code. It is reasonable because dynamic analysis can capture the runtime information, which can facilitate the understanding of current app operations. However, it has two issues that impede its practical use: *low scalability* that makes it costly to detect a huge amount of apps, especially for the Android app stores; *high dependency* that makes it impossible to deploy it on real devices, as DATs usually rely on an in-depth instrumentation or modifications to Android OS.

3) The DR of Anti-virus: Due to the aggressiveness of the malware, we cannot conduct a large scale user study. We manage to have 16 volunteers install the client app on their devices. Before the experiments, they need to have at least one AMT installed on their device. We also assure them that the possible attack is just proof of concept (POC), e.g., leaking

[4]https://github.com/pjlantz/droidbox
[5]https://labs.mwrinfosecurity.com/tools/drozer/

TABLE IV: The detection results of AMTs on real devices, where column ✓ means "passed" and ✗ means "detected"

Phone Model	OS	SDK	AMTs	Inst.	Runt.	Succ.
Nexus S	3.0.1	11	McAfee	✓	✓	Y
Nexus 4	4.0.1	24	Bitdefender	✓	✓	Y
Nexus 5	5.0.1	21	360 Security	✗	✓	Y
Nexus 6P	6.0.1	23	360 Security	✓	✓	Y
Nexus 6P	6.0.1	23	Norton	✗	✓	Y
Samsung Note 3	5.0	21	Kaspersky	✓	✓	Y
Samsung Note 4	5.1.1	21	AVG	✓	✓	Y
Samsung Galaxy 4	4.4.2	19	Lookout	✓	✓	Y
Samsung Galaxy 5	4.4.2	19	CleanMaster	✓	✓	Y
Samsung Galaxy 6	5.0.2	21	AVG	✓	✓	Y
Huawei P8	5.0.1	21	AntiVirus	✓	✓	Y
Huawei Honor 7	5.0.2	21	Avast	✗	✓	Y
Nexus 6P	6.0.1	23	Avast	✓	✓	Y
Asus Zendfon Selfie	5.0.2	21	None	✓	✓	Y
Xiaomi MI 2	5.0.2	21	Avira	✓	✓	Y
Xiaomi Note 2	5.0	21	Baidu	✓	✓	N

IMEI, leaking number of contacts, leaking a file's name and size only, and deleting the copied one of a user file. We replace the code of aggressive AFs (e.g., encryption) with that for POC. The profiles of devices and the detection results are presented in Table IV. Attack vectors for each device are selected by *LP-based AF selection* module. Details can be found at [36].

Evasiveness of malware. Generally, MYSTIQUE-S can easily bypass the scanning of most of AMTs shown in Table IV. Column *Inst.* means the scanning results of AMTs just after installation; column *Runt.* means whether AMTs give alerts when the attack is in progress; column *Succ.* means whether attacks succeed on the device.

As the attack is conducted by dynamically loading malicious code from the remote server and executing it locally, most AMTs fail to identify the maliciousness of client app after installation. There are only three AMTs that report the installed app as suspicious — 360 SECURITY, AVAST and NORTON.

Interestingly, in Table IV, the client app passes the scanning of 360 SECURITY on Nexus 6P, while it is detected by 360 SECURITY on Nexus 5. The detection capability in latest Android OS is even degraded in some cases. We speculate that some AMTs such as 360 SECURITY requests *root* permission to perform an in-depth scanning. So they even exploit n-day or zero-day vulnerabilities for rooting the user device. However, the latest Android OS (i.e., 6.0) fixes all known vulnerabilities and increases the difficulty in rooting. In reality, this weakens the detection capabilities of these AMTs. In addition, NORTON reports our client app as suspicious. In further testing, we find that NORTON also reports many commonly used apps (which are normally regarded as benign) as suspicious, e.g., Facebook, GrabTaxi and Line. The reason is that NORTON employs a strict detection mechanism that gives many false positives. Note for the three alerted cases by AMTs, the attacks still succeed.

No matter whether AMTs give alerts after installation or at runtime, we confirm the attack results by checking whether the honeypot (§ VII-A) receives the attack response. We find the attack succeed on 15 out of 16 devices, while fails on Xiaomi Note 2. Further inspection shows this Xiaomi phone has compatibility problem with the client app that causes the failure of attacks.

Transparency of malware. We collect the feedback of user experiences from the 16 volunteers. They cannot notice the malicious behaviors of the client app, without any obvious symptom (e.g., high network traffic and high CPU consumption) observed. Hence, MYSTIQUE-S can silently conduct the malicious behaviors specified by the remote server while causing no attention of users. We attribute this to the adoption of LP-based AF selection for different user scenarios, which optimizes between the number of selected AFs, the chance to be detected, and the latency (overheads) of the attack.

C. RQ3: Generating Recent Attacks in Real Cases

To show the adaptivity of our tool, we combine the AFs to constitute the recent popular real-world attacks on Android.

1) Hacking Online Banking: Recently, numerous customers of Australia's largest banks are the victims of a sophisticated Android attack that steals banking details and thwarts two-factor authentication security. Our running example originates from this attack. Customers of mobile banking apps are at risk from the malware, which hides on infected devices waiting until users open legitimate banking apps. The malware then superimposes a fake login GUI over the top for intercepting usernames and passwords. The malware can mimic up to 20 mobile banking apps from Australia, New Zealand and Turkey, as well as login GUIs for PayPal, eBay, WhatsApp and etc.

Attack prerequisites. The following conditions need to be satisfied before attacking: **p1**, the specified malware is installed and started on victims' devices; **p2**, the malware is granted with sufficient permissions, including android.permission.INTERNET and android.permission.RECEIVE_SMS; **p3**, the banking app employs the mechanism of two-factor authentication which needs to send verification code to the register phone. The client app of MYSTIQUE-S can ride on some benign apps using "repackaging" [48]. In § VII-B, we show that the client app can easily evade the detection by AMTs, which guarantees **p1**. To satisfy **p2**, the client app asks for the necessary permissions (defined in the *manifest* file), which can be granted at installation (before Android 6.0) or at runtime (since Android 6.0). To satisfy **p3**, we mimic the login GUIs of the banking apps, such as CitiBank.

Attack vector. According to user's installed mobile banking apps (e.g., CitiBank), the user-tailored AFs (including the phishing feature for CitiBank login GUI) are selected. The service provider then generates the malicious payloads that consist of malicious code and commands to execute. The malicious code can be referred to Fig. 7, and the commands in BDL can be referred to § VI-B.

Damage of attack. We have distributed this attack to 5 Android phones, from Android 4.0 to Android 6.0, and successfully collected the credentials and two-factor authentication. We discuss the possible damage from two aspects: the value of attack target and the user awareness of the attack. Once the bank account has been hacked, the attacker can obtain direct benefits from the victim which can cause a huge damage to the victims. From the perspective of users, there is no perceivable difference between the benign and phishing app, as Android activity as well as the views on it provide almost no hints for manual authentication. Unlike the phishing website that uses the fake URLs, careful users can spot some hints to authenticate. Therefore, it easily escapes from the awareness of victims.

2) Extortion app — Simplocker: Since the extortion malware *Simplocker* was found in 2014, ransomware has been swarming into the mobile app stores [49]. After launch, *Simplocker* starts to encrypt files in a background thread. The encrypted files can be any format, and the encryption is by AES cipher. However, the encryption key is hard-coded in the binary file, which can be used to decrypt the files. It is believed that *Simplocker* is just a proof-of-concept or an early development version of more severe and complicated variants of ransomware.

Attack prerequisites. The attack needs to meet such prerequisites: **p1**, the malware is installed and started on user devices; **p2**, the malware is granted with sufficient permissions, e.g., the permission (android.permission.WRITE_EXTERNAL_STORAGE) to access to the storage. The same to the first case, MYSTIQUE-S satisfies **p1** and **p2**.

Attack vector. After installed, MYSTIQUE-S collects the information of the user device. If many important files are found on the device (e.g., many new taken photos or created user files), the user-tailored AFs (e.g., encryption, deletion) are selected. As the BDL below, four AFs are selected for this attack, and there are three constraints for these four features. Normally, the permission android.permission.INTERNET is acquired by default, which ensures the downloading of malicious payload. The features are deployed into the main thread of the daemon service, which can be represented as INTENT_SERVICE::MAIN.

Features:
encryption, deletion, android.permission.WRITE_EXTERNAL_STORAGE
android.permission.INTERNET (for downloading payload)
Constraints:
$encryption \wedge deletion \Leftrightarrow extortion$
$encryption \Rightarrow android.permission.WRITE\_EXTERNAL\_STORAGE$
$deletion \Rightarrow android.permission.WRITE\_EXTERNAL\_STORAGE$
BDL:
$INTENT\_SERVICE :: MAIN :: ENCRYPT(CIPHER, FOLDER)$
$\rightarrow INTENT\_SERVICE :: MAIN :: DELETE(FOLDER)$

Execution of the payloads generated from the BDL above performs the *encryption* on a certain folder, and deletes it.

Damage of attack. This attack is distributed via MYSTIQUE-S, which is started by the client app. In this experiment, we use the AES to encrypt the specify folder and then delete the original files. The extortion attack can severely damage users' information properties. The target files, which are encrypted with a unknown cipher, may be very important to the victims. In addition, the extortion attack can optionally have the AF *sink*, if the user device has 4G connection. This operation can further cause the leak of users' privacy.

Spamming: $INTENT\_SERVICE :: MAIN :: SINK(SMS, LOCAL\_VARIABLE)$
$(\rightarrow INTENT\_SERVICE :: MAIN :: SINK(SMS, LOCAL\_VARIABLE))*$
Privacy: $INTENT\_SERVICE :: MAIN :: SOURCE(CONTACT :: CONTACT)$
$\rightarrow INTENT\_SERVICE :: MAIN :: SINK(HTTP, LOCAL\_VARIABLE)$
Privilege escalation: $INTENT\_SERVICE :: MAIN :: RUN(SHELL)$

3) Miscellaneousness: MYSTIQUE-S can easily configure and generate a variety of attacks. For example, spamming is the kind of attacks which is annoying and exhaustive in recent years [50]. This attack can be easily achieved by frequently conducting *sink* operation. Hence, the SMS spamming can be represented with the BDL as above. MYSTIQUE-S can easily deploy the attack of privacy leakage using various source-sink patterns, which involve 11 types of sensitive information such as contact and SMS (Full details of sensitive information can be referred to [36]). As the above BDL, the client app can obtain the contact information on the current device. In addition, MYSTIQUE-S can be further used to launch the attack of privilege escalation which needs shell code to root the device.

VIII. DISCUSSION

A. Threats to validity

The internal threats to validity of evaluation stem from three aspects. First, regarding the completeness of attacks considered in this study, we just focus on the four types of attacks (§ II-B) at this stage. In future, we will consider attacks such as privilege escalation that roots the device via vulnerability exploitation. Supporting privilege escalation will make MYSTIQUE-S similar to METASPLOIT on Android. Second, for the three goals of malware generation (§ V-B), aggressiveness and detectability are security related, but latency is more on quality of service (QoS). In future, we will consider other security or QoS related goals, e.g., to minimize the communication times and data size to exchange between the server and the client app. Last, for the values of d_i and l_i of a feature (§ V-B), we now manually define these values according to our understanding of these attacks and results reported by the study [11]. Accord to our preliminary study on different values of d_i and l_i, we find the impact of values (d_i and l_i) for AFs is minor to the results of feature selection, compared with the constraints among AFs. As the variant features to be selected for one common AF is usually less than 5, the optimal set of AFs to be returned is often similar to an near-optimal set. A further empirical study is required for better setup of d_i and l_i for different attacks.

The external threats are mainly two-fold. First, the malware samples for FODA are mostly from GENOME and DREBIN. Both of them contain many out-of-date malware, due to the everlasting malware evolution and creation. To ensure the timeliness of the FM of malware, we have considered some recent samples of attacks of information leakage and extortion (§ VII-C). Another threat is about the availability of real devices and AMTs. More real devices need to be tested with more various AMTs.

B. To be or not be obfuscated?

In this study, we do not further adopt the possible obfuscation techniques for the client app or the generated malicious code. Owing to the low detectability that we observed in the experiments (§ VII), it is not necessary to use extra obfuscation techniques for evading AMT detection. We also observe that existing AMTs do not sufficiently check the data that is received by a client app from the remote server at runtime. The rationale is that performing such check would impose a heavy burden on the performance. Besides, applying no obfuscation techniques eases the manual check of the generated malicious code for the experts. In reality, bytecode obfuscation techniques [2, 3] or wrapping payloads into native dynamic-link library (DLL) are applied for malware.

C. Possible enhancements for existing AMTs

To detect malware generated by MYSTIQUE-S, we propose three different solutions, which are discussed as follows:

1) Detecting C&C Communications Between the Client App and the Service Provider: Actually, the first solution is usually used for botnet or intrusion detection, but not a standard feature of AMTs. We find that AMTs normally cannot afford to check the data exchange of each app on Android. Firewalls often adopt the network traffic or DNS analysis [51, 52] to detect the C&C communication. Considering our tool as a testing framework rather than a real attack tool, we do not encode the C&C communication or use proxy strategies to prevent the tracing of the service provider. So detecting and hiding C&C communication is a topic different from this paper.

2) Detecting Dynamic Code Loading by Hybrid Analysis: Hybrid Analysis (i.e., integrating static and dynamic analysis) can help identify our malware. The first step is to conduct static analysis on Android apps to find those that employ dynamic loading techniques (e.g., by checking the existence of DEXCLASSLOADER). Nevertheless, using dynamic loading techniques does not imply that the app is malicious, as many benign apps employ dynamic loading for unnoticed update [14, 53, 54]. Then, we need to build a white list for trusted apps and server IP domains that are relevant to dynamic code loading. Last, for the app on the white list, we still need to have dynamic analysis in order to verify the benignity of the downloaded code or file at runtime. The study [14] refers to the work on downloaded file check at runtime on android.

3) Detecting Attacks by Realtime Monitoring and Security Verification: The above two solutions are to check the communication manners and dynamic code loading mechanism, which may not sufficiently prove the maliciousness of an app. Thus, the last solution is to have runtime anomaly detection. We have witnessed the effectiveness of realtime monitoring in [11] to detect the malware of privacy leakage. However, it encounters many issues when it deals with dynamically loaded malware. Most of realtime monitoring is based on information flow analysis, and therefore, the incompleteness of sensitive information to be monitored can cause insufficient detection. Moreover, information flow based detection mainly targets malware of privacy leakage, while missing malware of other attacks (e.g., ransomware). We propose to have some sandbox [55] or instrumentation mechanism (e.g., ARTIST [56]) to monitor the behaviors of an app with dynamically loaded code: checking entities it accesses, alerting users about suspicious changes to apps or system files, etc. Besides, information obtained at runtime should be verified against the system security properties and requirements [57], e.g., 1). no app should request the GPS location, and later send it out via the Internet (possibly to transmit the stolen location information); 2). no two apps should be able to have collusion attack (app *a* requests the GPS location, app *b* gets the information by IPC with *a*, and app *b* sends it out via the Internet).

IX. RELATED WORK

AMT auditing. ANDROTOTAL [58] is an integrated framework to automatically test the detection capabilities of anti-virus tools. Christodorescu and Jha [59] leverage four types of obfuscation techniques to test the capabilities of commercial anti-virus tools. ADAM [8] employs several transformation techniques to generate polymorphic malware, and test 10 prestigious anti-virus tools. DROIDCHAMELEON [2, 3] collects three types of transformation attacks in Android, and the authors have used these attacks to audit the AMTs. Huang *et al.* [17] assess the detection capabilities of 30 top anti-virus tools from two aspects: malware scanning and engine updating. The study [60] also reports that existing AMTs are suspectable to dynamically loaded malware, using the existing malware.

Among the above studies, studies [58, 60] aim to provide the platform to automate the process of AMT auditing. Currently, for offline detection of AMTs, we run some scripts to audit the AMTs via the service of VIRUSTOTAL. For the runtime detection in § VII-B, the users manually check AMTs' report about the running apps. ADAM [8], DROIDCHAMELEON [2, 3] utilize the evasion techniques (e.g., obfuscation, repacking, transformation attacks) to generate malware variants for AMT auditing. Apparently, evasion techniques generate no new valid malware, but variants with the same malicious intent. In contrast, our study facilitates creating new malware via combinations of various modularized AFs and evasion techniques.

Regarding to the advance of runtime based AMT evasion attacks, Huang *et al.* identify the Android stroke vulnerability (ASV) of system service [61] and the weakness of AMTs at time points of scanning and engine update [17]. In this study, as using new system vulnerability (e.g., ASV) or AMT weakness certainly fails the AMTs, we just modularize and then combine the AFs of existing GENOME malware for generating new malware. In this manner, we audit AMTs. Note that MYSTIQUE-S can easily add new AFs that are modularized from the malicious code of vulnerability exploits (e.g., that of ASV). However, such AFs might be too advanced for the purpose of AMT auditing, but useful for the recurrence of an attack.

Automated malware creation. According to the report by Zhou *et al.* [6], 86% of the 1260 samples are repackaged versions of benign apps with malicious payloads. Hence, repacking benign apps with malicious payloads is a cheap and fast way of malware creation. Based on the extra modules introduced by malicious payloads, the approach of module decoupling can effectively detect repackaged malware [48]. Recently, genetic programming has been applied to create malware in an automated way and evade the detection [9, 10]. Cani *et. al.* [10] employ μGP to automatically create new malware that is undetectable for AMTs, and inject malicious code into a benign app to construct a Trojan horse. Aydogan and Sen [9] also adopt genetic programming to create Android malware. Different from the mutation operations on instructions of executables [10], Aydogan *et al.* mutate the CFGs (control flow graphs) that are extracted from smali code of GENOME malware [6]. Their experiments show that the new generated malware can easily bypass the detection of AMTs. As shown in the study [10], mutating malware faces one critical problem: deciding whether a mutant still retains the characteristics of malware is a major issue of the evaluator. Compared with these mutation-based approaches, our approach evolves existing malware via combinations of the modularized AFs, which easily guarantee the maliciousness of new malware.

Evasive malware generation. Our work is also related to the generation of evasive or dynamically loaded Android malware.

To evade the detection of AMTs [2, 3], DROIDCHAMELEON integrates three types of transformation techniques and generates obfuscated Android malware. Some evasion techniques used in DROIDCHAMELEON [2, 3] are identified as evasion features by Meng *et al.* in [11]. Hence, for the malware that contains malicious payloads at compile time before execution, the obfuscation [62] or evasion techniques (i.e., [2, 3]) are very useful in failing the detection of AMTs.

Maier *et al.* [63] propose SAND-FINGER to construct the *divide-and-conquer* attack, which fingerprints the characteristics of popular sandboxes and decides to (or not to) load malicious code at runtime. SAND-FINGER's basic idea is to load the malicious code at runtime when the sandbox is not detected by scanning the fingerprinted characteristics. Unlike our approach, SAND-FINGER does not modularize AFs. Instead, it divides a malware sample into benign and malicious part. Essentially, to prevent from detection, Maier *et al.* propose the evasion features of sandbox fingerprints. It is interesting to make MYSTIQUE-S adopt these features, and check how these features can help evade the detection mechanism proposed in § VIII-C. In addition, Petsas *et al.* [64] propose three heuristics (static heuristics, dynamic heuristics and hypervisor heuristics) to fail dynamic analysis of Android malware. According to results of checking heuristics rules, the attack decides whether to launch the malicious payloads at run-time. In contrast, our malicious payloads are delivered from the remote server at runtime and can be purged after execution.

Dynamic code loading, as a code updating technique on its own, is not harmful. Previous study [14] reports that 9.25% of 1632 popular apps dynamically load external code. According to Aysan *et al.* [54], 19.60% of 25,000 apps from three markets datasets make use this technique for updating purpose. According to the recent empirical study by Maier *et al.* [65], among 14,885 malicious and 22,032 benign apps, 36.4% of malicious samples and 13.1% of benign apps use dynamic code loading. Hence, dynamic code loading is becoming an important evasion feature for Android malware. Based on the findings in [65] and our observations in this paper, the protection from attacks with this technique is still unsatisfactory for existing AMTs. Last, our study is different from the empirical study [65] as below. Maier *et al.* focus on dynamic code (and script) loading, and investigate how it relates to malware [65] and how it can be addressed. Our study focus on combining dynamic code loading with different modularized AFs, and investigate the capability of existing AMTs.

X. CONCLUSION

In this paper, we propose to adopt the SPLE in order to modularize the common attack behaviors and construct the corresponding conceptual model (i.e., the FM) for android malware. To provide a benchmark for dynamically loaded malicious code, MYSTIQUE-S adopts the DSPL techniques and makes the attack as a service, which facilitates the integration with other tools for AMT audit and penetration testing. We also evaluate the effectiveness of MYSTIQUE-S and the evasiveness of the generated malicious code on 16 real devices with 4 different recent attacks. In future, we will investigate the effectiveness of other attack and evasion features, such as obfuscating the generated malicious code. In addition, MYSTIQUE-S enables many studies on the malware generation and AMT auditing. Lastly, we will investigate the detection strategies for the malware generated by our tool on the fly.

ACKNOWLEDGMENT

This research is supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] AV-TEST, "AV-TEST Product Review and Certification Report-May/2016," https://www.av-test.org/en/antivirus/mobile-devices/android/may-2016/.
[2] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks," in *AsiaCCS*, 2013, pp. 329–334.
[3] ——, "Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.
[4] 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.
[5] H. Gunadi and A. Tiu, "Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System," *CoRR*, vol. abs/1311.2362, 2013.
[6] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *IEEE S&P*, 2012, pp. 95–109.
[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] 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*, 2013, pp. 82–101.
[9] E. Aydogan and S. Sen, "Automatic Generation of Mobile Malwares Using Genetic Programming," in *Applications of Evolutionary Computation*, vol. 9028, 2015.
[10] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda, "Towards Automated Malware Creation: Code Generation and Code Integration," in *SAC*, 2014, pp. 157–160.
[11] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving Android Malware for Auditing Anti-Malware Tools," in *AsiaCCS*, 2016.
[12] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.
[13] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, vol. 19, no. 4, 2002.
[14] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *NDSS'14*, 2014.
[15] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel, "Tailoring Dynamic Software Product Lines," in *GPCE*, 2011, pp. 3–12.
[16] D. A. Mundie and D. M. McIntire, "An Ontology for Malware Analysis," in *ARES*, 2013, pp. 556–558.
[17] 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*, 2015, pp. 7–18.
[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Tech. Rep., Nov 1990.
[19] D. S. Batory, "Feature Models, Grammars, and Propositional Formulas," in *SPLC*, 2005, pp. 7–20.

[20] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search based software engineering for software product line engineering: a survey and directions for future work," in *SPLC*, 2014, pp. 5–18.

[21] C. A. Castillo, "Android Malware Past, Present, and Future," Tech. Rep., 2012.

[22] T. Inc., "A Brief History of Mobile Malware," Tech. Rep., 2012.

[23] Symantec, "Internet Security Threat Report," Tech. Rep., 2016.

[24] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar, "New Privacy Issues in Mobile Telephony: Fix and Verification," in *CCS*, 2012, pp. 205–216.

[25] Y. Zhou and X. Jiang, "An Analysis of the AnserverBot Trojan," Tech. Rep., 2011. [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf

[26] McAfee Inc., "Mobile Threat Report: What's on the Horizon for 2016," Tech. Rep., 2016.

[27] Joji Hamada, "Simplocker: First Confirmed Ransomware for Android," http://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android.

[28] J. Crussell, C. Gibler, and H. Chen, "Attack of the Clones: Detecting Cloned Applications on Android Markets," in *ESORICS*, 2012, vol. 7459, pp. 37–54.

[29] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, "Detecting android malware using clone detection," *J. Comput. Sci. Technol.*, vol. 30, no. 5, pp. 942–956, 2015.

[30] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 306–317.

[31] M. Rangwala, P. Zhang, X. Zou, and F. Li, "A Taxonomy of Privilege Escalation Attacks in Android Applications," *Int. J. Secur. Netw.*, vol. 9, no. 1, pp. 40–55, Feb. 2014.

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

[33] 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, pp. 1043–1054.

[34] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *CCS*, 2012, pp. 217–228.

[35] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," in *NDSS*, 2015.

[36] "Mystique — Evolving Android Malware for Auditing Anti-Malware Tools," https://sites.google.com/site/malwareevolution/.

[37] 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.

[38] V. Avdiienko, K. Kuznetsov, A. Gorla, and A. Zeller, "Mining Apps for Abnormal Usage of Sensitive Data," in *ICSE*, 2015.

[39] A. Turner, "Malware hijacks big four australian banks' apps, steals two-factor sms codes," https://t.co/ud5P7C8Zzq, 2016.

[40] E. International, "ECMAScript 2015 Language Specification," Tech. Rep., 2015.

[41] P. van den Broek, "Optimization of product instantiation using integer programming," in *SPLC*, 2010, pp. 107–112.

[42] H. Ishibuchi, N. Tsukamoto, and Y. Nojima, "Evolutionary Many-Objective Optimization: A Short Review," in *CEC*, 2008, pp. 2419–2426.

[43] 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*, 2013, pp. 492–501.

[44] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon, "Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines," in *ICSE*, 2015.

[45] "Xposed Module Repository," http://repo.xposed.info/, 2016.

[46] "Cplex," http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud, 2016.

[47] 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*, 2010, pp. 1–6.

[48] 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*, 2013, pp. 185–196.

[49] ESET, "The Rise of Android Ransomware," Tech. Rep., 2014.

[50] TechEye, "Android malware MisoSMS one of the largest botnets to date," http://www.tgdaily.com/security-brief/83076-android-malware-misosms-one-of-the-largest-botnets-to-date.

[51] A. Zand, G. Vigna, X. Yan, and C. Kruegel, "Extracting probable command and control signatures for detecting botnets," in *SAC*, 2014, pp. 1657–1662.

[52] S. García, A. Zunino, and M. Campo, "Survey on network-based botnet detection methods," *Security and Communication Networks*, vol. 7, no. 5, pp. 878–903, 2014.

[53] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer, "Andrubis – 1,000,000 apps later: A view on current android malware behaviors," in *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014, pp. 3–17.

[54] A. I. Aysan and S. Sen, ""do you want to install an update of this application?" A rigorous analysis of updated android applications," in *IEEE 2nd International Conference on Cyber Security and Cloud Computing, CSCloud 2015, New York, NY, USA, November 3-5, 2015*, 2015, pp. 181–186.

[55] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "Baredroid: Large-scale analysis of android apps on real devices," in *ACSAC*, 2015, pp. 71–80.

[56] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," *CoRR*, vol. abs/1607.06619, 2016.

[57] A. Bauer, J. Küster, and G. Vegliach, "Runtime verification meets android security," in *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, 2012, pp. 174–180.

[58] F. Maggi, A. Valdi, and S. Zanero, "AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors," in *SPSM*, 2013, pp. 49–54.

[59] M. Christodorescu and S. Jha, "Testing Malware Detectors," in *ISSTA*, 2004, pp. 34–44.

[60] R. Fedler, M. Kulicke, and J. Schütte, "An Antivirus API for Android Malware Recognition," in *MALWARE*, 2013, pp. 77–84.

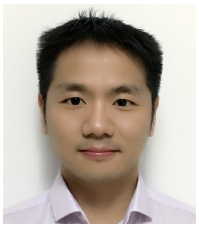
[61] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, 2015, pp. 1236–1247.

[62] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware," *Comput. Secur.*, vol. 51, 2015.

[63] D. Maier, T. Müller, and M. Protsenko, "Divide-and-Conquer: Why Android Malware cannot be stopped," in *ARES*.

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

[65] D. Maier, M. Protsenko, and T. Müller, "A game of droid and mouse: The threat of split-personality malware on android," *Computers & Security*, vol. 54, pp. 2–15, 2015.



Yinxing Xue is currently a Research Scientist with Nanyang Technological University (NTU). He received the B.E. and M.E. degree from Wuhan University, China. In 2013, he received the Ph.D. degree in computer science from National University of Singapore (NUS). Since Feb. 2013, he had been a Research Scientist with Temasek Laboratories, NUS. Since Jan. 2015, he has been a Research Scientist with Temasek Laboratories, NTU. His research interest includes software program analysis, software product line engineering, cyber security issues (e.g., malware detection, intrusion detection and vulnerability detection).

Guozhu Meng received Bachelor and Master degree in school of computer science and technology from Tianjin University, China in 2009 and 2012. He worked in Temasek lab in National University of Singapore for one year as an Associate Scientist. Since 2013, he started his Ph.D study in school of computer science and engineering of Nanyang Technology University, Singapore. His research interests include mobile security, software engineering and program analysis.

Yang Liu received the bachelors degree in computing and the Ph.D. degree from the National University of Singapore (NUS), in 2005 and 2010, respectively. He continued with his postdoctoral work in NUS. Since 2012, he has been with Nanyang Technological University, as an Assistant Professor. His research focuses on software engineering, formal methods, and security. In particular, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the art model checker, Process Analysis Toolkit.

Tian Huat Tan is currently a senior researcher at Acronis. Tian Huat completed his Ph.D. at the School of Computing, National University of Singapore. He has worked at Singapore University of Technology and Design(SUTD) as a research fellow. His research interests include artificial intelligent, cyber-security, and system verification.

Hongxu Chen is a PhD student in Nanyang Technological University. He received Bachelor of Science degree in Nanjing University of Science and Technology in 2011 and Master of Computer Science degree in Shanghai Jiaotong University in 2014 respectively. Hongxu's research interests include program language theories, cyber-security, program analysis, and software engineering.

Jun Sun is currently an Associate Professor at Singapore University of Technology and Design (SUTD). He received Bachelor and PhD degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member of SUTD since 2010. He was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, formal methods, program analysis and cyber-security. He is the co-founder of the PAT model checker.

Jie Zhang is an Associate Professor of the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He is also an Academic Fellow of the Institute of Asian Consumer Insight and Associate of the Singapore Institute of Manufacturing Technology (SIMTech). He obtained Ph.D. in Cheriton School of Computer Science from University of Waterloo, Canada, in 2009. During PhD study, he held the prestigious NSERC Alexander Graham Bell Canada Graduate Scholarship rewarded for top PhD students across Canada. He was also the recipient of the Alumni Gold Medal at the 2009 Convocation Ceremony. The Gold Medal is awarded once a year to honour the top PhD graduate from the University of Waterloo. His papers have been published by top journals and conferences and won several best paper awards. Jie Zhang is also active in serving research communities.