Semantic Modelling of Android Malware for Effective Malware Comprehension, Detection, and Classification

Guozhu Meng*, Yinxing Xue*, Zhengzi Xu*,
Yang Liu*, Jie Zhang*, and Annamalai Narayanan†
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore

ABSTRACT

Malware has posed a major threat to the Android ecosystem. Existing malware detection tools mainly rely on signature- or feature- based approaches, failing to provide detailed information beyond the mere detection. In this work, we propose a precise semantic model of Android malware based on Deterministic Symbolic Automaton (DSA) for the purpose of malware comprehension, detection and classification. It shows that DSA can capture the common malicious behaviors of a malware family, as well as the malware variants. Based on DSA, we develop an automatic analysis framework, named SMART, which learns DSA by detecting and summarizing semantic clones from malware families, and then extracts semantic features from the learned DSA to classify malware according to the attack patterns. We conduct the experiments in both malware benchmark and 223,170 real-world apps. The results show that SMART builds meaningful semantic models and outperforms both state-of-the-art approaches and anti-virus tools in malware detection. SMART identifies 4583 new malware in real-world apps that are missed by most anti-virus tools. The classification step further identifies new malware variants and unknown families.

CCS Concepts

•Security and privacy → Malware and its mitigation; •Software and its engineering → Abstraction, modeling and modularity; •Theory of computation → *Program analysis;*

Keywords

Android Malware Detection, Malware Modelling, Clone Detection, Determined Symbolic Automata

1. INTRODUCTION

Android has become a popular mobile operating system installed on 1 billion of devices, which accounts for more than 81.5% of all smartphones in 2014 [48]. Its prevalence and the prosperity of Android marketplaces have made it a hot target for attackers to upload various malware. The total number of mobile malware samples has increased by 112%, and exceeds 5 million [52] in 2014.

Existing approaches to detecting Android malware typically rely on bytecode similarity analysis [39, 71, 72], machine learning techniques [34, 21, 25, 67, 69] or information flow analysis [41, 43, 45, 26, 28, 30]. Bytecode similarity analysis usually adopts pair-wise comparison, with time complexity $O(n^2)$ for comparing n possible malware variants. Machine learning based approaches can be effective and efficient in detecting malware, but only provide some predictive features for malware without explaining the malicious behaviors involved. The approaches relying on information flow analysis are accurate, but not efficient for the large-scale detection.

The key in successful defense to malware variants or zero-day attacks is to understand the different attack behaviors in malware and prevent them in a proactive way. However, none of the aforementioned approaches provides a complete semantic explanation of attacks, the behaviors involved in attacks, and the actions inside behaviors.

In this paper, we propose an explicit semantic model to capture the malicious behaviors of a malware family so as to achieve a precise comprehension. We represent the essential elements of malicious behaviors in malware variants as Deterministic Symbolic Automata (DSA) [59] with transitions being system APIs (e.g., see Fig. 1). The motivation is that one malicious behavior may occur in a variety of manners, which can easily bypass malware detection tools based on malware signatures or patterns. For example, there are multiple ways of sending private data from mobile devices to a remote server, e.g., using an HttpURLConnection object to write a stream to the server, or launching a browser to attach the data in the URL. Although the two implementations differ notably, they fulfil the same task. DSA can encode alternative transitions among states, which enables capturing malware variants of the same kind.

The ultimate goal of our study is to build a semantic model of malware towards effective detection, classification, and forensics. To the best of our knowledge, it is the *first attempt* to propose a semantic model considering malware variants.

Based on the semantic model, we implement an automatic malware analysis framework, named **S**emantic **M**odelling of **A**nd**R**oid a**T**tack (SMART), which provides three key features—automatic semantic model construction from existing malware families, efficient malware detection and malware classification. The overall workflow of SMART is shown in Fig. 3. First, SMART starts with identifying semantic clones [44] among malware variants to capture the common

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931043

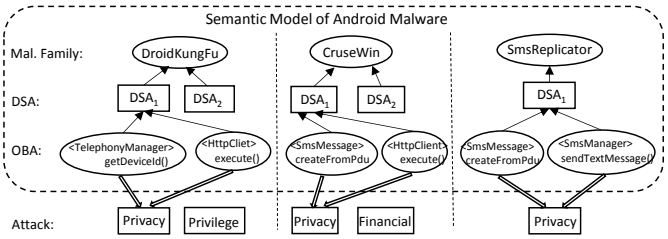

Figure 1: The hierarchy of the semantic model

malicious behaviors. We propose a bytecode clone detection method (§ 5.1) to effectively identify similar malicious code in the form of bytecode clones. To ensure the scalability and eliminate the noise of third party libraries, SMART filters out commonly-known libraries according to [24]. For each clone set, we introduce an efficient bytecode differencing algorithm to build a DSA by comparing n clone instances at one time with the complexity $O(n)$ (§ 5.2). From DSA, low level actions are extracted (§ 5.3). Finally, a semantic model in Fig. 1 is stored into our DSA database. DSA is able to cover most of the attack, but not for all. The applicability and generality of the DSA model is discussed in § 8.

Once the semantic models are constructed, we extract malicious features from DSA and perform malware detection via machine learning techniques (§ 6.1). Then, for each suspicious app, SMART performs a static analysis to check whether malicious behaviors from our DSA database is contained in the app via a DSA inclusion check (§ 6.2). If DSA inclusion succeeds, we can confirm the maliciousness of the app and malware families they belong to. Otherwise, we identify the actions that are contained in the app for the sake of understanding its behaviors. The DSA-based semantic model facilitates finding new malware variants, new families or even new attacks. The machine learning and the static analysis complement each other to achieve both good performance and high accuracy. In summary, we make the following contributions:

- We propose a hierarchical semantic model of Android malware based on DSA, which links malware family, malicious behaviors and fine-grained actions. To the best of our knowledge, this is the first work in this direction.
- We propose an effective method to extract the common malicious behaviors as DSA from malware variants of an Android malware family.
- We combine machine learning and static analysis to detect and classify malware. The first phase quickly identifies suspicious apps, and the second phase uses DSA inclusion to further confirm and classify the suspicious apps. Hence we can achieve both scalability and accuracy.
- We test SMART on 5560 known malware and 223,170 real-world apps. The results show that SMART outperforms many anti-virus (AV) tools and academic approaches, with a precision of 86.7% for malware detection. For wild prediction, it detects 4583 new malware variants.

2. RELATED WORK

There are substantial number of related work on android malware analysis and also a comprehensive survey [55]. Here we list some most relevant work due to space limitation.

Bytecode similarity analysis. As malicious repackaged apps usually share the common basic functions and features, the idea of DroidMOSS [72] relies on the pair-wise check of the similarity between two apps and uses it as the basis to detect repackaged apps. DroidMOSS proposes to generate fingerprint via a fuzzy hashing technique to localize and identify the code changes due to repackaging behaviors. Similarly, DNADroid [39] pair-wisely measures the similarity between two apps to find the cloned attacks. Similarity score is calculated by applying the VF2 algorithm [38] to compute *subgraph isomorphisms* existing in program dependency graphs (PDGs) between methods in candidate apps. To overcome the scalability limitation due to pair-wise comparison, PiggyApp [71] and WuKong [64] adopt the clustering technique to address the piggybacking and clone detection among similar mobile apps.

Mining or learning malicious behaviors. A precise model of malicious behaviors can significantly improve the accuracy of detection. Previous efforts have been made to distinguish malicious apps from benign ones by classification. Crowdroid [34] uses Android API call sequences in Linux kernel as features. Drebin [25] adopts static analysis to extract features relevant to malicious attacks, e.g., permissions and API calls. DroidAPIMiner [21] extracts features from dangerous Android APIs calls, APIs parameters and package level information for the classification. MAST [35] selects permissions, Intent filters, the existence of native code and zip files, then applies *Multiple Correspondence Analysis* on Genome malware collection. Peiravian *et. al.* [58] take permissions and API calls as features to train a model to detect malware. DroidMiner [67] proposes a two-tiered behavior graph to model malicious program logic into a vector of threat modalities, and then applies classification according to these modalities. DroidSIFT [69] further models API-relevant behaviors into weighted contextual API dependency graphs and classifies malware based on these graphs. Recently, AppContext [68] proposes to differentiating malicious and benign behaviors based on the contexts: *events* and *conditions* that trigger security-sensitive behaviors. Predictive features are further extracted from the differentiating results.

There also exist some works on malware model using automata. Babić *et. al.* [29] use *tree automata* to model malicious behaviors in malware. They use the extracted system call dataflow dependency graphs to infer k-testable tree automata. Preda *et. al.* [60] propose *abstract symbolic automata (ASA)* to present the syntactic and semantics of binary executables. Aiming at identifying the messages between malware and the environment, Bonfante *et. al.* [32] conduct a dynamic analysis on malware, and employ automata to descriminate types of different execution by an arbitrary message. SMART is the first attempt to use automata to describe a whole malware family, with describing the common and different parts across the malware variants.

Malware detection with other techniques. The traditional malware detection is based on signature or hashing matching [46, 36], which is incapable of addressing the obfuscation and malware variants. Information flow analysis is another effective approach in malware detection, which usually adopts dynamic or static taint analysis for tracking information-flow in mobile apps. For instance, TaintDroid [41] and VetDroid [70] use dynamic taint analysis by instrumenting the Dalvik VM, while Apposcopy [43] and [45, 26] employ static taint analysis for scalability. In Apposcopy, results of static taint analysis are combined with high level signatures of malware to further speed up the detection. Recently, MUDFLOW [28] leverages static taint

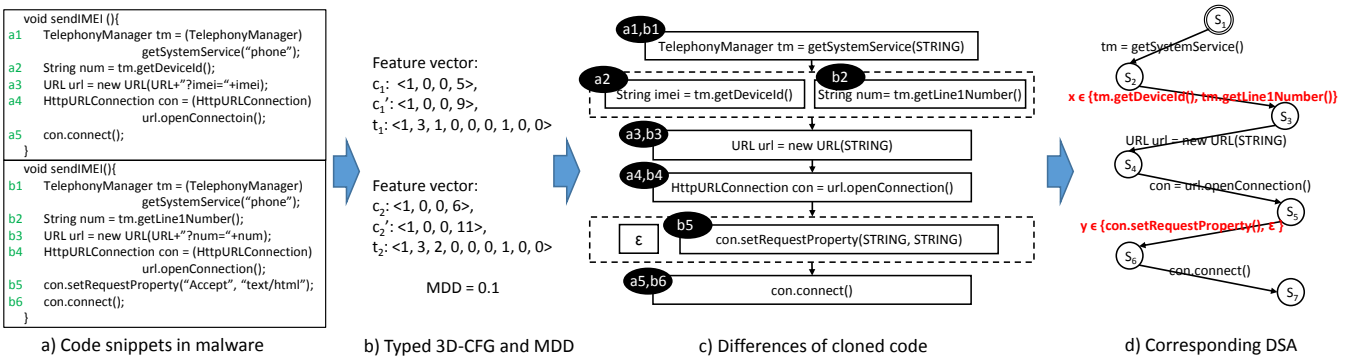

Figure 2: A running example of representing malicious behavior in DSA for two samples from DroidKungFu

analysis to detect information leaks via comparing source-to-sink patterns in malware or benignware, and massively mines app repositories for patterns of *normal* data flow. The mined patterns are used as predictive features for detection.

Different from the previous works, we emphasise the malware model considering the possible variants. Our proposed semantic model, which captures commonality and variety of malicious behaviors, helps to understand the essence of the attack behind. Our combined approach of malware detection has advances in resisting both obfuscation and variants of malware, and overcomes the challenges in scalability.

3. MODELLING ANDROID MALWARE

In the program comprehension domain, *deterministic finite automaton (DFA)* has been used to model the program logic of a method in Android app [23, 66].

DFA can depict a concrete malicious behavior exactly, but it is inflexible in presenting a behavior with many variants. Thus, inspired by the fuzzy Android code search engine CODOTA [56], we use DSA [59] to model the program logic of one malicious behavior with variants.

DEFINITION 1. *A deterministic symbolic automaton (DSA) is a 6-tuple* $(\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F}, Vars)$ *where* $\mathcal{Q}$ *is a finite set of states;* Σ *is a finite alphabet, indicating executed statements in a malicious behavior;* $Vars$ *is another finite alphabet, and it is defined as a set of variables which can be assigned with arbitrary statements. In addition, the symbol* $\epsilon \in Vars$ *denotes a null statement;* δ *is hereby a revised transition function:* $\mathcal{Q} \times (\Sigma \cup Vars) \rightarrow \mathcal{Q}$*, representing a transition relation; and* $\mathcal{F} \subseteq \mathcal{Q}$ *is a set of final states.*

Compared with DFA, DSA is equipped with the capability of expressing variant events via symbolic variables. $Vars$ in a DSA is a unique feature that covers the variety of transitions amongst nodes in the DSA. Similarly, $traces(DSA)$ is the set of all execution paths $\langle s_0, s_1, s_2, ..., s_n \rangle$ contained in the DSA, where $s_0 = q_0$, $s_i \xrightarrow{e_i} s_{i+1}$, $e_i \in \Sigma$ and $s_n \in \mathcal{F}$. Malicious apps in the same malware family may have slightly different behaviors, resulting in a set of code clones that contains one clone instance from each malware variant in the family. Therefore, each clone set can be summarized as one DSA, with $\mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$ denoting the common parts and $\mathcal{Q} \times Vars \rightarrow \mathcal{Q}$ denoting differences among clone instances.

For a malware family with many variants, we can model the malicious behaviors using a number of DSA, where each DSA is summarized from a set of cloned methods in these variants (§ 5). These DSA together can model the malicious behaviors shared by these variants, i.e., the signature to fingerprint this malware family. However, one DSA describes a behavior involving a series of operations on multiple targets or resources. To have a fine-grained model of the operations at implementation level, we propose the concept of Object-based Actions (OBAs) (§ 5.3.2) to describe low-level atomic actions relevant to a certain resource in Android OS context.

Based on DSA and OBA, we propose a hierarchical model with three layers: 1) the top layer is the malware family model, which summarizes the common behaviors of malware families. It is modelled by a set of DSA. 2) the middle layer is an abstract model of a type of malicious behaviors along with its possible variants. DSA is the representation of malicious behaviors. 3) the bottom layer models the concrete objected-based actions (OBAs)—a partial DSA relevant to a target or resource object for composing a malicious behavior, e.g., deleting a system file or register entry.

Fig. 1 shows the hierarchical relationship of malware family, DSA, OBA and type of attacks. For instance, DroidKungFu [50] contains two types of malicious behaviors (DSA): privacy leakage and privilege escalation. The privacy leakage attack involves two OBAs: operations relevant to class TelephonyManager and HttpClient. The families CruseWin and SmsReplicator also contain the issue of privacy leakage, while involving different actions. Note that different OBAs can compose a specific type of attack, as the object or the resource under the OBAs decides the nature of the attack.

In our semantic model, we ignore the connections among DSA (or OBAs), which are data flow or control flow, in the part of malware family (or attack type). As the implementations of connections can vary greatly (e.g., data flow among DSA may use inter-procedural or external channels), we omit connections among DSA (or OBAs) for scalability issues.

Running Example. We select two samples from malware family DroidKungFu to show a concrete semantic model and how it is learned. One malicious behavior of DroidKungFu is to steal sensitive information (e.g., IMEI code and phone number), and send the information to a remote server. Fig. 2 shows how the malicious behavior is expressed in three representations: typed 3D-CFG used for fast clone detection, code statements with summarized differences, and finally a DSA used as its behavior model.

In Fig. 2(a), the common malicious behavior consists of two main steps: 1) obtain the private data (e.g., the IMEI code or phone number) — call the instance of TelephonyManager and invoke its method to access telephone data; 2) leak the data — create an instance of HttpURLConnection and send the data via this instance. Despite sharing the same malicious intent, two samples in Fig. 2(a) are different in implementation: 1)

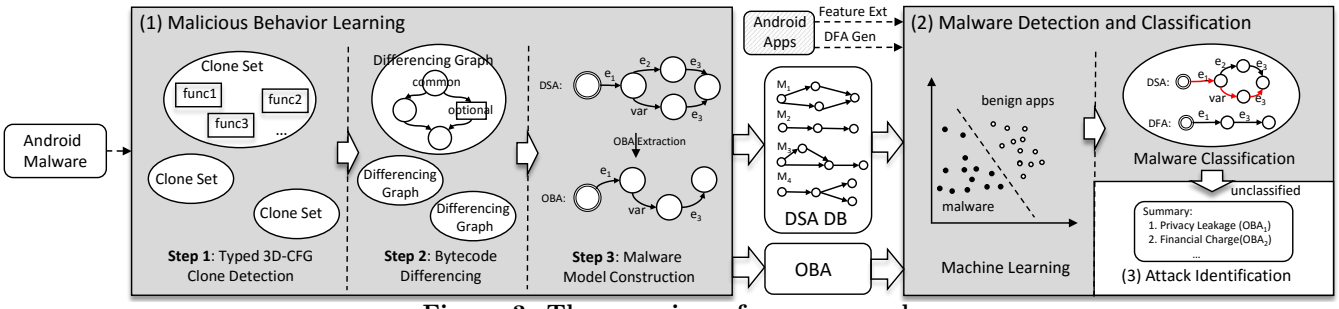

Figure 3: The overview of our approach

the first one fetches the IMEI code via getDeviceId, and the second one gets the phone number via getLine1Number; 2) the second one additionally invokes setRequetProperty to set up parameters for the connection.

Instead of modelling two different variants with two separate DFA or one combined DFA, we represent the two variants of malicious behavior of stealing information as a DSA, shown in Fig. 2(d). The *init* state of the constructed DSA has a transition which invokes the method getSystemService. The DSA proceeds in a subsequent transition which supports a variable $x \in \{\text{getDeviceId}, \text{getLine1Number}\}$. Then, the state proceeds with another variable $y \in \{\epsilon, \text{setRequestProperty}\}$, where ϵ denotes a null event for the transition. Thus, y denotes an optional operation. By virtue of variables x and y, this DSA precisely models four different implementations of behaviors (i.e., four valid variants). Thus this DSA can be used SMART for detection of variants of malicious behaviors and malware family classification. Furthermore, by analyzing OBAs in malware, we can identify the potential attacks that a suspicious app launches. For example, a contained OBA (e.g., relevant to TelephonyManager) can be used to identify a potential information leakage — the class TelephonyManager contains many methods (e.g., getDeviceId and getLine1Number) that allow attackers to stealthily obtain sensitive phone data.

4. THE SMART FRAMEWORK

SMART performs the modelling of malicious behaviors of existing Android malware, and uses these abstract semantic models to detect unknown malware or new variants. In the modelling part, the input is the bytecode of malware variants, and the output is the constructed DSA. In the detection part, the input is the DSA and a suspicious app, and the output is the detection and classification result. Fig. 3 shows the system architecture of SMART, which relies on static analysis of bytecode and proceeds in two phases:

- **Malicious behavior learning.** This step is based on bytecode clone detection and code differencing analysis. First, we perform bytecode clone detection to identify security-related clones from the malware variants in a malware family. Our bytecode clone detection is designed to tolerate slight differences among clone instances. Second, we adopt a code differencing analysis to learn both the common and optional parts of one type of malicious behaviors. Then we represent them using a unified statement graphs with summarized differences in dashed rectangles, as shown in Fig. 2(c). Last, we generate DSA from differencing graph and extract OBAs from DSA based on consecutive transitions of certain objects.
- **Malware detection and classification.** We propose a combined approach for malware detection to achieve efficiency and accuracy, simultaneously. First, we extract predictive features of attacks from the summarized DSA. With these features, we can identify suspicious apps via ML classification. Second, we perform a static analysis process, called DSA inclusion check, to determine if the concrete behavior model of an Android app (in the form of DFA) is semantically included in any learned DSA. A match between the DFA of the suspicious app and any DSA suggests that this app has the malicious behavior. If this app matches the majority of DSA of a certain malware family, SMART suggests this app is a variant of that family. If the DFA of this app fails to match any DSA, we extract the OBAs from its DFA, and compare them with those in existing DSA in order to identify the similar attack at a fine-grained level.

5. LEARNING MALICIOUS BEHAVIORS

5.1 Bytecode Clone Detection

3D-CFG [36, 37] is a structural feature, and used to measure the similarity between methods in/among Android apps. The basic idea in [36] is to assign a structural 3D-coordinate value for each node in the control flow graph (CFG) of a method, then calculate the mass centroid of these coordinates as the signature of the method. By checking the distance between two methods' centroids, it is decided whether they are clones.

DEFINITION 2. *For a given method, each node in the CFG is a basic code block. Each node has unique coordinates, denoted as a 3-tuple (x, y, z) [36], where x is its sequential number in the CFG; y is the number of its outgoing edges (control flow in CFG), and z is the loop depth of the node.*

3D-CFG with type information.

We propose a type-enriched 3D-CFG, which takes into account the type information of statements as a part of method signature to measure the similarity of two clone methods.

DEFINITION 3. [36] *Given a method m, the method signature $\overrightarrow{m} = (\overrightarrow{c_m}, \overrightarrow{c_m}', \overrightarrow{t_m})$ is a feature vector containing the structural and type information of the method, where $\overrightarrow{c_m}$ is the centroid, $\overrightarrow{c_m}'$ is the weighted centroid, and $\overrightarrow{t_m}$ is a vector in which t_i depicts the number of occurrences of i-th type of statements in this method.*

The centroid $\overrightarrow{c_m}$ of a method m is a 4-tuple (c_x, c_y, c_z, w):

- $w = \sum_{e(p,q) \in 3D-CFG} (w_p + w_q)$,
- $c_i = \frac{\sum_{e(p,q) \in 3D-CFG} (w_p i_p + w_q i_q)}{w}$, *where* $i \in \{x, y, z\}$

```
1 String m1(){                    1 String m2(){
2     TM tm=getSysSer();          2     String res=str.rep();
3     String id=tm.getId();       3     rep.rep("suffix");
4     return id;                  4     return res;
5 }                               5 }
```

Figure 4: A false positive using the 3D-CFG approach

where $e(p,q)$ is the edge between the node p and q, (x_p, y_p, z_p) is the coordinate of the node p, and w_p is the number of statements in the node.

To emphasize the importance of invocation statements, the weighted centroid is defined as $\overrightarrow{c_m}' = (c'_x, c'_y, c'_z, w')$ for method m, where $w' = w+N$, and N is the number of invoked statements; c'_x, c'_y and c'_z are recalculated according to the new weight w', respectively. For example, for the first method in Fig. 2(a), the centroid is $(1,0,0,5)$ and the weighted centroid is $(1,0,0,9)$ (only four invocation statements). Note that w' is 9, the sum of w and the number of invocation statements.

The difference between the centroids of two methods m_1 and m_2 is the primary condition to evaluate their similarity:

Definition 4. [36] *The Centroid Difference Degree (CDD) of two centroids* $\overrightarrow{c_1} = (c_{1x}, c_{1y}, c_{1z}, w_1)$, $\overrightarrow{c_2} = (c_{2x}, c_{2y}, c_{2z}, w_2)$ *(similarly for the weighted centroids) is*

$$CDD(\overrightarrow{c_1}, \overrightarrow{c_2}) = max(\frac{|c_{1x}-c_{2x}|}{c_{1x}+c_{2x}}, \frac{|c_{1y}-c_{2y}|}{c_{1y}+c_{2y}}, \frac{|c_{1z}-c_{2z}|}{c_{1z}+c_{2z}}, \frac{|w_1-w_2|}{w_1+w_2})$$

Given $\overrightarrow{t_1}$ and $\overrightarrow{t_2}$ that represent the vector of occurrences of statements with different types in method m_1 and m_2, respectively, *Type Difference Degree (TDD)* is defined as:

$$TDD(\overrightarrow{t_1}, \overrightarrow{t_2}) = \sum(\frac{|t_{1i}-t_{2i}|}{t_{1i}+t_{2i}})/|\{i|t_{1i} \neq 0 || t_{2i} \neq 0\}|$$

Given two methods, the method-level difference degree is defined as the maximum value of their centroid distance (CDD), their weight centroid distance (weighted CDD) and also the Type Difference Degree (TDD):

Definition 5. *Method Difference Degree (MDD) of two methods* $\overrightarrow{m_1} = (\overrightarrow{c_1}, \overrightarrow{c_1}', \overrightarrow{t_1})$ *and* $\overrightarrow{m_2} = (\overrightarrow{c_2}, \overrightarrow{c_2}', \overrightarrow{t_2})$ *is*

$$MDD(\overrightarrow{m_1}, \overrightarrow{m_2}) = max(CDD(\overrightarrow{c_1}, \overrightarrow{c_2}), CDD(\overrightarrow{c_1}', \overrightarrow{c_2}'), TDD(\overrightarrow{t_1}, \overrightarrow{t_2}))$$

Benefits of typed 3D-CFG. For the two methods in Fig. 4, the CDD for their centroids and weighted centroids are both 0 — they have only one path, both represented with a single block. Thus, they have the exactly same centroid, and will be clustered into the same clone set. The 3D-CFG method will treat them as clones. In our typed 3D-CFG, MDD is 5/12, as TDD is 5/12. If the threshold for MDD is empirically set to a small constant (e.g., 0.1), they are not clones.

For our example in Fig. 2(a), the CDD between $\overrightarrow{c_1}$ and $\overrightarrow{c_2}$, i.e., the maximum distance on the elements of 4-tuple, is calculated to be 1/13 and $CDD(\overrightarrow{c_1}', \overrightarrow{c_2}')$ to be 1/11, as shown in Fig. 2(b). To calculate $TDD(\overrightarrow{t_1}, \overrightarrow{t_2})$, we measure the average distance of the statement types that appear in m_1 or m_2 (we select 9 out of 15 types of statements defined in Soot [63] in Section 5.2). Since type vector $\overrightarrow{t_1}$ is $\langle 1,3,1,0,0,0,1,0,0\rangle$ and $\overrightarrow{t_2}$ is $\langle 1,3,2,0,0,0,1,0,0\rangle$, $TDD(\overrightarrow{t_1}, \overrightarrow{t_2}) = 1/12$. Thus, MDD for the example in Fig. 2(b) is 1/11.

Pre-filtering of non-malicious clones. To assure that the clones are related to malicious behaviors, we filter out non-malicious behaviors in three ways: 1) maintain a white list of common third-party libraries not to be considered in clone detection step. It includes 89 ads libraries, 18 social sdks, and 201 development tools [24]; 42.4% of malware samples include these common libraries. Thus, this step greatly improves the soundness of malware model learning. 2) set a threshold (denoted as θ_1) to retain the clones relevant to most variants in a family, since not all variants share the same clone set (possibly containing common malicious behaviors). For example, 11 out of 14 (78.6%) variants of family Zitmo send users' incoming SMS messages to attackers via HTTP, while the others via SMS. For the clone set sending privacy via HTTP, if the ratio of the number of its clone instances to the total number of variants in the family (78.6%) exceeds a predefined threshold (e.g., $\theta_1 = 75\%$) [51], we regard it as being attack related. 3) verify the clones according to the knowledge of malicious code. For samples in Genome, package and class name of malicious code can be identified.

Table 1: Elements to compare for different statements

Statement	Type	Elements to compare
$a = new\ Class()$	IDENTITY	TYPE(a), TYPE(Class)
$a = expr$	ASSIGN	TYPE(a), TYPE(expr)
$invoke\ count(a)$	INVOKE	NAME(count), TYPE(a)
$if(a > b)$	IF-ELSE	TYPE(a|b), TYPE(>)
goto a	GOTO	
return	RETURNVOID	
switch(a)	SWITCH	TYPE(a)
return a	RETURN	TYPE(a)
throw e	THROW	TYPE(e)

5.2 Bytecode Differencing

After clone detection, we can get a number of clusters of similar code at method level. We summarize them by identifying the common and also different parts of these cloned methods with a bytecode differencing algorithm. The basic idea of the differencing algorithm is borrowed from the concept of progressive alignment [42] in multiple DNA sequence alignment. Instead of using pair-wise comparison with complexity of $O(n^2)$, we compare n cloned methods in $n-1$ times. Specifically, we first compare two methods with the least MDD and get results in the form of a differencing graph (e.g., Fig. 2(c)). Then, the other $n-2$ ones are gradually compared to the intermediate differencing graph. Finally, a differencing graph unifying the commonality and variability among cloned methods is built.

Algorithm 1 depicts the bytecode differencing step. The input is a set of CFG G of the methods in a clone set. Given a CFG g, $v(g)$ is the set of vertex of g; $children(g,n)$ denotes the successors of the node n in g. The algorithm returns a differencing graph G_r, which aggregates the input graphs with matched common nodes and summarized optional nodes. First, we sequentialize the statements of each CFG at bytecode level via a preorder traversal (implemented using BodyExtractorWalker in Soot) (line 1); then we employ *longest common subsequence (LCS)* [54] to calculate common statements shared by these cloned methods (line 2). For example in Fig. 2 (c), running LCS for these two methods gives $commonTokens = \{(a_1,b_1),(a_3,b_3),(a_4,b_4),(a_5,b_6)\}$. Here, the matched statements n_1, n_2, ..., to n_i from different methods are denoted as the equivalent class $\{n_1,\ldots,n_i\}$. LCS at line 2 is adopted in [54] and [49] for source code differencing, while we adopt it for bytecode differencing.

In calculating LCS, a typed approach is used to determine the equivalence of two statements, We consider 9 basic types as listed in Table 1. Other statement types like NOP and RET are omitted as they do not carry any semantic meaning. Most types in Table 1 only use type information for comparison, except that the type INVOKE that needs textual matching for the name of invoked method. The rationale is that most textual information like customized variable or method

Algorithm 1: Bytecode Differencing

Input: G is a set of CFGs of cloned methods
Output: G_r is a differencing graph

```
1  list⟨list⟨Stmt⟩⟩ tokens := preorder traversal(G);
2  list⟨set⟨Stmt⟩⟩ commonTokens := LCS(tokens);
3  let v(G_r) ← ∅ be the set of vertex in G_r;
4  for set⟨Stmt⟩ node ∈ commonTokens do
5      v(G_r) = v(G_r) ∪ node;
6  for g ∈ G do
7      for src ∈ v(g) do
8          List⟨Node⟩ dstList := children(g, src);
9          Node nSrc = findNode(G_r, src);
10         for dst ∈ dstList do
11             Node nDst = findNode(G_r, dst);
12             G_r.addEdge(nSrc, nDst);
13 return G_r;
```

names are lost after compilation and obfuscation. Only the full method names of Android APIs are kept. In Fig. 2 (c), statement pairs (a_1, b_1), (a_3, b_3), (a_4, b_4) and (a_5, b_6) are *identical* since they invoke the same Android APIs with the same parameter type *String*, regardless of the *String* value. (a_2, b_2) is not identical as they invoke different Android APIs. Additionally, b_5 is optional and unmatched to any statement in the first method, denoted as (ϵ, b_5).

Lines 6-12 in Algorithm 1 add the original control flow relationship between statements existing in G to the corresponding statements in graph G_r. Finally, in G_r the vertex is either identical (common amongst cloned methods), or optional (different amongst cloned methods). G_r retains the original control flow relationships. At line 9, $findNode(G_r, src)$ is to find the node in G_r which contains src. As shown in Fig. 2(c), the control flow relationship between (a_1, b_1) and (a_2, b_2) is added back. Similarly, all the other relationships between statements in G can be recovered.

5.3 Semantic Model Construction

5.3.1 DSA Construction

After obtaining differencing graphs for summarized cloned methods, we construct the corresponding DSA. The conversion from a differencing graph to a DSA is a *line-digraph* problem that has been nicely addressed in [47]. Statements with summarized differences are represented with variables in DSA. In Fig. 2(c), the matched statement pair (a_2, b_2), in which a_2 and b_2 are not identical, can be represented as variable x—invoking either getDeviceId or getLine1Number. As there is no matched statement in the first method for b_5, variable y is used to denote either an optional invocation of setRequestProperty or *null* operation (ϵ).

To capture the essence of malware, we apply two filters to remove the statements regardless of their summarized differences during the construction of DSA. First, we filter out the statements that contain no invocation or branch information, such as int a=10 and return a. However, the statements with branch information like if (a>10) will be kept. The rationale to remove these statements is that—they have no direct interaction with the Android system or the external environment, which means that it can only operate its enclosed data and internal logic in apps, but not directly impact on Android and the external environment [69]. Second, we filter out the statements invoking methods that are not Android APIs. Thus, invocations of methods from third-party libraries are pruned. As the detection is Android-API name sensitive (§ 6) and methods from third party libraries are easily obfuscated (e.g., Proguard [53]), we keep invocations to Android APIs that cannot be easily altered.

Algorithm 2: Objected-based Action Extraction

Input: D is a DSA
Output: $OBAs$ is a list of OBA, initially empty.

```
1  for node ∈ D do
2      for initialized object obj ∈ node do
3          partial DSA d = forward_slicing(obj);
4          OBA = {d_s | node ∈ d ∧ node.contains(obj)};
5          OBAs = OBAs ∪ OBA;
6  return OBAs;
```

5.3.2 Object-based Action Extraction

To provide a fine-grained model for malicious behaviors, we perform an *OBA extraction* on the DSA. One malicious behavior can be split into several atomic actions, and any two actions may have data flow dependency in between.

As illustrated in Fig. 2(d), the behaviors represented by the DSA invoke 5 or 6 operations. Intuitively, these operations can be roughly grouped into two OBAs with meaningful semantics—*obtain sensitive information* and *send out the information*. Specifically, the former action is realized via calling the method getDeviceId or getLine1Number in class TelephonyManager; the latter calls a sequence of methods ⟨openConnection, setRequestProperty$^+$, connect⟩ in class HttpURLConnection. For these API usage patterns related to a specific object, we call them object-based actions (OBAs). In addition, to obtain a complete lifecycle for an object, we consolidate the creation and invocations of this object. In Fig. 2 (a), statement a_1 returns the global instance of TelephonyManager. Combining the creation (a_1) and the invocation (a_2) of the object, we obtain a complete OBA for TelephonyManager.

Algorithm 2 depicts the process to extract actions from DSA. First, we traverse the nodes contained in DSA (line 1). OBAs usually begin with an initialization, i.e., the aforementioned creation of an object, which can return an object of a certain class (line 2). For example, the statement tm=getSystemService() in our running example. Then, we use *forward slicing* [31] to get a partial DSA of which nodes contain the object (lines 3, 4). For example, the invocation of the method getDeviceId in TelephonyManager is categorized into TelehonyManager-based actions. Last, Algorithm 2 returns a list of OBAs, e.g., two OBAs (one TelephonyManager-based and one HTTP-based) are returned for the running example.

6. MALWARE DETECTION AND CLASSIFICATION

The learned DSA can be utilized as signatures (in the form of directed graphs) for malware detection and classification. However, the direct searching based on DSA matching is not scalable [69]. In this section, we propose a combined approach that fast filters out benign apps via machine learning, and then conducts DSA inclusion check on the suspicious apps.

6.1 Machine Learning Based Detection

To enable fast malware detection, a machine learning classifier is trained to detect malware. Particularly, two types of features we use for machine learning are listed as follows:

Feature used in previous studies. Basically, features derived from *Risky Android API* (denoted as **feature set**

F1) and *Bigram call sequence of risky APIs* (denoted as ***F2***) are widely used in existing malware detection on Android.

Risky Android APIs refer to APIs with high risks if not properly used by users. Usually, the invocations of these risky Android APIs potentially lead to certain malicious behaviors without the awareness of users. We select 469 risky Android APIs [27] that are frequently used in malware, e.g., getLine1Number to access phone number and sendTextMessage to send an SMS message. In Table 2, 189 out of 469 Android APIs appear in the malware code and constitute *F1*.

Bigram call sequence of risky APIs refer to 2-length call patterns of risky APIs that highly likely appear in malicious behaviours. For example, malware Zitmo and benign app Wechat both register a BroadcastReceiver to receive incoming SMS messages. Then Zitmo sends messages to a remote server, while Wechat only shows messages to the local user. Inspecting call sequences of APIs provides more information on the intention of behaviors, and distinguishes malicious apps from benign ones. Hence, we perform an inter-procedural analysis to extract bigram call sequence as *F2*. In Table 2, out of all bigram call sequences of the 189 risky APIs, 5923 bigram features appear in malware code and constitute *F2*.

Selecting features according to DSA. To guarantee the comprehensiveness of features, we first add more APIs into *F1* – 182 APIs which do not acquire permissions but are commonly used in malware (e.g., getContentResolver), 147 APIs that are related to Java *reflection*, and 2 APIs that are used to invoke native code. Second, we extract more abundant bigram call sequences of APIs. However, using all features of APIs and bigram call sequences leads to the problem of *curse of dimensionality* [10]. To mitigate this problem, we select APIs and bigram call sequences existing in DSA as ***F3***. In Table 2, compared with *F1*&*F2*, *F3* contains the features and bigram features of the 182 extra APIs that we consider. Note that each feature in *F3* is required to be contained in at least one DSA.

After specifying feature vectors, we apply machine learning to train a classifier which can distinguish malicious apps from benign ones. We employ the *Random Forest* classifier [33] (it achieves the best classification result based on our experiments in Section 7.2) to gain a well-trained model from the training set. According to classification results, we can determine if one candidate app is benign or suspicious. For the suspicious apps, we will conduct the DSA based detection to confirm, classify them into known malware families, and identify the attacks involved.

6.2 DSA based Detection and Classification

The list of suspicious apps resulting from machine learning step contains false positives, which requires further scrutiny. Thus, to confirm these suspicious apps, we propose the DSA based static analysis, which proceeds in three phases: 1) *DFA construction*. For a given Android app, we analyse and construct one DFA for each method contained in this app. As shown in Fig. 5, method getImei in this app is converted into a DFA, illustrating the control flow of the method. In this way, we can obtain a set of DFA for an app; 2) *Inclusion querying*. The extracted DFA will be sent to inquiry the DSA database. In this paper, we perform a DSA inclusion algorithm to determine if any DSA of malware includes this DFA. If one of the DFA in the app is identified as being included in any DSA in the malware family A, it is called a

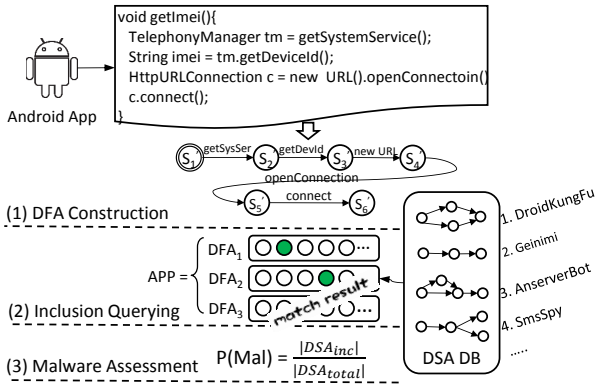

Figure 5: The process of DSA based detection

match to A; 3) *Malware assessment*. We identify the number of matches as well as its proportion for each malware family, and determine that the app is malicious if the proportion is larger than a certain threshold (denoted as θ_2).

DSA inclusion is to check if all accepted paths in a given DFA are also accepted in the DSA. We define it as follows:

Definition 6. *A DFA dfa is included in a DSA dsa* ***iff*** $traces(dfa) \subseteq traces(dsa)$.

Given one DSA, the inclusion check answers whether all paths in the DFA can be found in the DSA or not. We adopt the anti-chain algorithm [40, 65] to check whether a DFA is included in the DSA (more details in [18]). For a method from the suspicious app in the previous step, if the DFA in Fig. 5 fails to match any learned DSA, it appears that the suspicious app may not reuse the existing attack behaviors or fall into an existing malware family. For further scrutiny, we check if the DFA complies with one or multiple OBAs via OBA inclusion (i.e., checking whether the DFA is included in the OBA, as an OBA is a partial DSA). Consequently, we identify the attacks that highly co-occur with the matched OBAs. In Fig. 5, the DFA matches two kinds of OBAs, i.e., TelephonyManager-based action to access IMEI code and HTTP-based action to transfer information, suggesting a potential *privacy leakage* attack.

7. EVALUATION

SMART is implemented in Java with 10K+ LOC. The experiments are conducted on a Ubuntu 14.04 desktop with Intel Xeon(R) CPU E5-16500 and 16G Memory. Details on SMART and experimental data are available in [18]. The data sets used in our experiments are as follows:

- **Malware benchmark for training (D1)**. We use the latest Drebin [25] malware collection containing 179 malware families and 5,560 malware apps. Note that Drebin includes the famous malware collection Genome that contains 49 malware families and 1,260 samples.
- **Real-world Android apps (D2)**. We crawled totally 223,170 apps from Google Play and 16 popular third party marketplaces. These marketplaces are deployed either in US or China. These apps are collected from 2013 to 2016.
- **Benign apps for training (D3)**. We select 5,600 apps in Google Play, on which 99.8% of available apps are benign according to [74]. We verify the benignity of these apps based on the report of VirusTotal [19] and collect 5,560 benign ones.

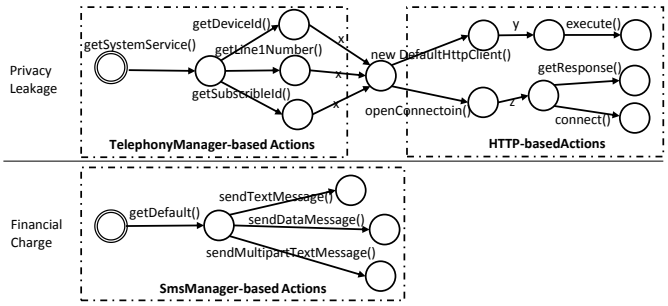

Figure 6: The DSA and actions for common attacks

Note that $D1\&D3$ constitute the labelled training dataset used in Table 2 and Table 3; $D2$ is the unlabelled dataset used for wild prediction on unlabelled apps in Table 4. The experiments are conducted to evaluate our approach in terms of five aspects explained in the subsections.

7.1 RQ1: Evaluation of the Semantic Model

Controlled experiments on the data set of DREBIN [25] are conducted to evaluate the usefulness of our semantic models. For each malware family, we generate a set of DSA and extract the corresponding OBAs to characterize attack behaviors. On average, there are 101 DSA learned for each family, which denotes the average complexity of attacks involved. There are around 20 nodes (i.e., code statements) on average in one DSA, containing about 6 variables that occupy 30% of all nodes in one DSA. In addition, we identify 20 types of OBAs with 193 variants existing in DREBIN. See [18] for detailed models.

Soundness & completeness. To justify the soundness of the DSA models, we select a number of samples for each malware family in DREBIN. We omit the families with only one sample, as a DSA generated from one sample is identical as the corresponding DFA. We find that all of the selected samples are correctly detected as malware. It proves that the constructed DSA are able to capture the essential parts of known malware. With an in-depth inspection of constructed DSA of 179 malware families, we confirm that the set of DSA for each malware family contains at least one malicious behavior. This proves the completeness of learned DSA—each malware family has malicious behaviors modelled as DSA. Moreover, due to the filtering mechanisms in bytecode clone detection in Section 5.1, we may omit a few malicious behaviors existing in the minority of variants. However, the common malicious behaviors are retained in the form of DSA.

As the OBAs are extracted from DSA and related to the "assets" users possess, they are capable of composing attacks. The two examples of attacks in Fig. 6 demonstrate the soundness of OBAs. The attack *privacy leakage* contains two basic operations: access sensitive information and send out the information. According to the result of OBA extraction, we find that the attack of stealing phone's profile can be implemented via TelephonyManager-based and HTTP-based actions. Similarly, the attack *financial charge* only involves one type of actions, i.e., SmsManager-based actions. As 20 types of OBAs are verified to be related to the assets exploitable by attacks, the OBAs cover the 6 common attacks (See [18]) in DREBIN, indicating the completeness of OBAs.

Observations of malware evolution. Since DSA is a comprehensive summarization of malware variants, we observe malware evolution from the generated DSA. Specifically, two ways of malware evolution or mutations to variant are observed: 1) In many malware families, variants are evolving in non-functional content, such as parameters (e.g., the address of remote server and the number of receiving phones) — malware variants of *parametrised clones*; 2) Variants in some malware families are enhanced by supporting more similar operations — malware variants of *gapped clones*. For example, Zsone registers a BroadcastReceiver to monitor the reception of SMS, and cancels the broadcast if the message comes from premium numbers. Meanwhile, it keeps adding new premium numbers in its variants. Most of this type of variants can be captured by our clone detection.

Summary. DSA can effectively model and help to detect the malware variants with exact clones, parameterised clones and clones containing small gaps. However, in our evaluation, we find one FN case missed by SMART—replacing functional code by equivalent code (e.g., DroidKungFu uses Socket to replace DefaultHttpClient to perform the communication operation) or using advanced obfuscation (e.g., String encoding, reflection and native code).

Table 2: Accuracies of classifiers using different feature sets on the training dataset that contains Drebin

Feature Set	# Features	Precision	Recall	F-Measure
F1	189	92.4	92.3	92.3
F2	5923	95.2	95.0	95.0
F1 & F2	6112	95.4	95.3	95.3
F3	12514	97.0	97.0	97.0

7.2 RQ2: Malware Detection based on ML

In this experiment, we select 5,560 malware samples in DREBIN and 5,560 benign apps from Google Play as our data set, and evaluate the results of different classifiers and the enhancement via DSA of machine learning based malware detection. All experimental results are obtained by performing 10-fold cross validation (CV).

The accuracy of the detection classifiers. To evaluate the accuracy of our training model, we extract features *F3* (§ 6.1) from DSA in our data set. We compare these classifiers: *AdaBoost* (87.8%), *C4.5* (93.8%), *Linear SVM* (90.4%), *Naïve Bayes* (81.6%) and *Random Forest* (97.0%). It is concluded that Random Forest achieves the best classification result. Hence, we employ Random Forest to train our model.

DSA-enhanced classifier.

We set up a comprehensive experiment to investigate the usefulness of different feature sets and DSA in the classification. Table 2 shows that feature set *F3* achieves the best result with the Random Forest classifier. As introduced in Section 6.1, feature set *F1* derived from risky APIs is used in recent study [21]. Feature set *F2*, which also has features derived from bigram call sequences, is evaluated in the state-of-the-art tool [57]. Thus, feature set *F3* derived from our DSA is more effective in detecting malware than feature sets *F1*, *F2* and its combination. The rationale is that *F3* excludes many bigram call sequences that do not relate to malicious behaviors, and moreover, provides significant features from DSA such as reflection and invocation of native code.

Moreover, we list significant features which can effectively distinguish malware from benign apps in Table 3. The first two columns in Table 3 show the most remarkable features as well as their weight in the classification extracted from DSA. 3 out of top 5 features are related to the access to local storage,

which are commonly used in Android malware to dynamically load payload (e.g., BaseBridge), store sensitive information (e.g., DroidKungFu), which are rarely used in benign apps; and the second two columns show features extracted from the whole code. In these five remarkable features, bigram call sequences pause → start and setVideoPath → pause are related to the operation of video player in Android. Generally, they are not considered the components of malicious behaviors. As a result, the top features extracted from DSA are more reasonable and representative for malicious behaviors (see the full list of weights of features in [18]); the third two columns show the significantly different features and their difference between these two experiments.

Summary. In the 10-fold CV training, SMART achieves the highest accuracy (97%) when using *F3* with *Random Forest*, by virtue of the feature selection based on the DSA.

7.3 RQ3: Evaluation on Real World Apps

We collect 223,170 free Android apps from Google play and third-party Android markets. The data sets are listed in Table 4 with the corresponding total number, and the number of suspicious apps detected by SMART as well as prestigious anti-virus (AV) tools, including AVG (AG), Avast (AT), BitDefender (BD), F-Secure (FS) and Sophos (SO). As shown in Table 4, SMART can detect 4,583 (2.1%) out of all collected apps based on machine learning (ML), while anti-virus tools only regard totally 527 (2.5%) of the 4,583 as malware. With the DSA inclusion check (DSA), SMART rightly classifies 527 (out of 4,583) samples into 23 known malware families — this implies that the approach based on DSA has consistent results with that based on ML, while ML reports more suspicious ones. According to DSA inclusion check, about 2.5% of the malware are classified into existing malware families from DREBIN, while others are not due to the timeliness of DREBIN. Our training data set DREBIN is mainly collected from 2010 to 2013, while real-world apps are collected from Dec. 2013 to Jan. 2016. Plenty of known malware had been removed from each marketplace. Still, among the detected variants detected by DSA inclusion, we identify 22 variants of family Boxer and 12 variants of family Foncy, both of which send unauthorized SMS.

We perform attack identification for the suspicious apps (reported by ML) via extracting OBAs. We identify the 5 most popular actions employed by these apps, which are TelephonyManager-based actions (89.9%), HTTP-based actions (71.6%), ContentResolver-based actions (66.1%), AssetManager-based actions (53.9%), and SmsManager-based actions (40.0%). We have manually checked 100 apps, and 76% of them are found containing malicious behaviors, which are mainly privacy leakage and sending SMS message to premium numbers. Taking the app "Super SMS Quick Delete" from AppsApk for example, we extracted OBAs existing in this app, and found it had TelephonyManager-based actions, with which it can access IMEI, IMSI, etc. In addition, it had SmsManager-based actions which can send SMS messages via sendTextMessage, as well as HTTP-based actions. It is concluded that this app can expose sensitive information to attackers. All the analysis results can be found at [18].

Summary. The ML (using *F1*&*F3*) and DSA inclusion in SMART detect more malware variants than the AV tools. Compared with ML, DSA inclusion is susceptible to the timeliness of the training set for family identification. However, it is more tolerant of malware variants. In contrast, AV tools detect more malware while many of them (76.6%) belong to *Adware*, which are not considered by SMART.

7.4 RQ4: Resilience to Malware Variants

To validate the resilience of SMART to the transformation attacks [61] and advanced variants, we compare the detection rate of SMART with other AV tools on manually crafted variants of known malware. The malware variants are crafted in three ways: 1) employ transformation attacks [61] to generate malware variants for testing (variants 1-11); 2) remove the malicious code from malware to test the false positive rate (variant 12); 3) craft malicious apps in terms of DSA and actions summarized from the malware collection (variant 13). The variants are published at [18].

As shown in Table 5, SMART exhibits the highest detection rate, as our combined detection approach (ML+DSA) considers the essential part of malware that are unchanged in these variants. AV tools achieve overall 32.3% detection rate of these variants. Note that the tool SO, which detects 79 real-world malware, fails to detect any crafted variants. By careful inspection of its mobile app, we infer that it relies on exact matching of signature of malware. Besides, we learn three kinds of features utilized by AV tools as follows:

- **Non-code files**. Specifically, the file *AndroidManifest.xml*, which declares acquired permissions, activities, services, etc., is scanned. BITDEFENDER can resist two complex transformations (variants 8, 9) when retaining AndroidManifest.xml, but fail when removing it.
- **Structure of code files**. AV tools check the structure of classes and methods in an app. Thus, if we keep the structure of the original app, however, remove the malicious code inside, these AV tools produce false positives.
- **Lexical features of code**. We create a new variant (No. 13) by following the DSA in malware family SMSSpy, which differs from the original malware in lexical features. None of AV tools is capable of detecting it.

Summary. SMART is capable in identifying malware variants according to transformation attacks. Variants generated according to DSA, e.g., 4 variants derived from Figure 2(d), can bypass the above three strategies of AV tools.

7.5 RQ5: Scalability & Efficiency

We measure the performance for the two phases of SMART, the offline model learning, and online malware detection and classification.

Offline model learning mainly consists of two computations: 1) typed 3D-CFG generation and 2) clone differencing and DSA generation. Given categorized malware data, we only need to perform model learning once. For each family of malware in DREBIN, the average time for clone detection is 72.5s, and in total it takes around 3.6h for 179 families. The clone differencing and DSA generation needs 167.5s on average. Although it takes a long time (around 19 days) to generate all 18,000 DSA, it only needs to be done once and the computation can run in parallel using multiple machines.

Online malware detection consists of two steps: 1) ML based malware detection and 2) DSA based inclusion for family classification. To check if an app is malicious, the ML step takes 13.4s on average (13.3s for feature extraction and 0.1s for prediction). If the app is detected as malware, it takes 105.9s on average to perform DSA based classification

Table 3: Top features w/o DSA of machine learning

Features in DSA		Features in code		Most different features	
Features	Weight	Feature	Weight	Feature	Difference
getCacheDir	0.19	getDatabasePath	0.22	getCacheDir → getDatabasePath	-0.20
getContentResolver → openInputStream	0.13	getCacheDir → getDatabasePath	0.21	getDatabasePath	-0.18
getPackageManager → getContentResolver	0.13	getContentResolver → openInputStream	0.17	setVideoPath → pause	-0.16
openInputStream	0.10	pause → start	0.17	pause → start	0.16
openXmlResourceParser	0.07	setVideoPath → pause	0.16	getPackageManager → getInstallerPackageName	0.09

Table 4: The data sets for evaluation on real-world apps

Marketplace	# App	SMART		Anti-Virus Tools				
		ML	DSA	AG	AT	BD	FS	SO
AndroidDrawer [2]	11731	200	5	246	152	200	212	187
AnZhi [3]	46757	848	34	702	613	276	300	796
Apkmirror [7]	23441	165	11	201	180	79	165	88
AppsApk [1]	2481	86	26	40	23	8	12	43
ChinaMobile [8]	1714	61	10	384	118	98	102	443
Coolapk [9]	19969	452	25	402	241	40	80	242
Eoemarket [11]	5895	101	3	953	346	433	606	780
FDroid [4]	4533	139	29	10	8	3	5	3
Flyme [12]	10927	614	121	1281	122	305	488	366
GetJar [13]	42633	498	89	2322	211	205	1233	1055
GFan [14]	1000	75	7	89	65	70	62	54
Google Play [5]	7643	45	37	220	81	55	89	41
Huawei [15]	9856	275	49	1730	1128	526	679	828
SlideMe [6]	5770	63	8	866	523	325	124	324
Wandoujia [20]	3066	163	20	189	50	120	156	62
Wangyi [17]	7272	120	8	138	142	98	86	87
Xiaomi [16]	18482	678	45	2497	999	1005	1332	1539
Total	223170	4583	527	12270	5002	3846	5731	6938

Table 5: The capability of detecting malware variants

Malware Variants	SMART	Anti-Virus Tools				
		AG	AT	BD	FS	SO
1. Repacking	✓	✓	✗	✓	✓	✗
2. Dis- and Re-assembling	✓	✓	✗	✓	✓	✗
3. Changing Package Name	✓	✓	✗	✓	✓	✗
4. Identifier Renaming	✓	✓	✗	✓	✓	✗
5. Data Encryption	✓	✗	✗	✓	✓	✗
6. Call Indirections	✓	✗	✗	✓	✓	✗
7. Code Reordering	✓	✗	✗	✓	✓	✗
8. Junk Code Insertion	✓	✗	✗	✓	✗	✗
9. Function Out- and Inlining	✓	✗	✗	✓	✓	✗
10.Other Simple Transformations	✓	✗	✗	✗	✗	✗
11.Composite Transformations	✓	✗	✗	✗	✗	✗
12.Remove Malicious Code	✗	✓	✓	✓	✓	✓
13.Construction based on DSA	✓	✗	✗	✗	✗	✗
Detection Rate (%)	100	30.8	0	69.2	61.5	0

to confirm and classify it according to the 18,000 DSA in our database. Assuming the probability of an app detected by ML is 0.03 (§ 7.3), the expected time for the detection and classification is 16.6s for each app, which makes SMART efficient for a large-scale scan.

8. DISCUSSION

Threat to validity. Threats to *internal validity* come from two thresholds used in bytecode clone detection θ_1 in Section 5.1 and malware assessment θ_2 in Section 6.2. A high value of θ_1 leads to few accidental clones but also few valid clones that are partially shared by malware variants, and vice versa. θ_2 intuitively specifies the maximal tolerance to the mutation of malware. If θ_2 is high, the DSA based approach can achieve more accurate results while tolerate less mutations of variants, and vice versa. Threats to *external validity* are two-folds: the *timeliness* of malware dataset and the *deficiency* of static analysis in our machine learning approach. Since the new malware is emerging every day and our DSA based approach performs an accurate matching with known malware, it can degrade the result if apps contain new malware that is not included in the known malware collection; Similar to [28], we do not carry out a complete static analysis to elevate the performance in ML approach. Inherently, static analysis is susceptible to dynamic loading and reflection, which is another threat to external validity.

Sufficiency of DSA in modelling Android malware. According to [73], malware samples in the same family carry similar malicious behaviors, which lead to clones involving similar attack targets and manners. Hence, malicious behaviors can be well captured with clone detection, and DSA can present the essential parts of malicious behavior while remain the variety of malware variants. Second, Android attacks usually call the OS built-in API. Thus we model the API call as the transition in DSA. The new attack that dynamically loads malicious payloads may not have the oblivious behaviors or DSA [62], but it still relies on some reflection APIs to conduct loading. Last, to avoid the path explosion problem, condition guards (arguments of API invocation) are not considered in the behavior model of DSA or DFA. Similarly, existing work on Android API dependency graph does not consider invocation arguments.

Usefulness of DSA in malware detection. As DSA can capture the common malicious behaviors, we use the learned DSA to guide the feature extraction in DREBIN (i.e., only extracting the concerned features included in DSA). Even using thousands of extra features derived from the 182 extra APIs (§ 5.3), the ML training stage becomes more efficient with 10% reduction of running time yet without affecting the accuracy (both as high as 98%). Besides, compared with other tools, SMART has the advantage in finding fine-grained attack actions, representing them as OBAs based on analysis of source-sink path [28] and control flow graph [22].

Enhancement of ML based malware detection. Features of bigram call sequences of Android APIs, especially those contained in DSA, can improve the detection accuracy of the ML based approaches by 5%-10%. As bigram call sequences retain the direct relationship between APIs, it can effectively depict coarse grained malicious behaviors [69, 28]. We limit the n-gram analysis of call sequence to bigram due to the efficiency issue. Note that the results of evaluation of SMART on testing datasets show that the introduction of bigram features does not cause overfitting problem, as the FP (13.7%) rate of SMART is still low.

9. CONCLUSION

In the paper, we built a hierarchical semantic model for Android malware. We developed a framework, named SMART, to automatically learn models from malware, and use a combined approach of machine learning and DSA inclusion to detect and classify malware. Experiments show that our approach can achieve both efficiency and scalability.

10. ACKNOWLEDGMENTS

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

11. REFERENCES

[1] Android Apps, Download APK, Android Applications, Android APK. http://www.appsapk.com/, 2014.

[2] AndroidDrawer.com - APK Download of Free Android Apps. http://www.androiddrawer.com/, 2014.

[3] AnZhi. http://www.anzhi.com/, 2014.

[4] F-Droid: Free and Open Source Android App Repository. http://f-droid.org, 2014.

[5] Google Play. http://play.google.com/store/, 2014.

[6] SlideME | Android Apps Market: Download Free & Paid Android Applications. http://slideme.org/, 2014.

[7] APKMirror - #APKPLZ #SOONBACKANSWER. http://www.apkmirror.com/, 2015.

[8] ChinaMobile. http://mm.10086.cn/, 2015.

[9] Coolapk. http://www.coolapk.com/, 2015.

[10] Curse of dimensionality. http://en.wikipedia.org/wiki/Curse_of_dimensionality, 2015.

[11] EOEMarket. http://www.eoemarket.com/, 2015.

[12] Flyme. http://app.flyme.cn/, 2015.

[13] GetJar - Download Free Apps, Games and Themes APK. http://www.getjar.com/, 2015.

[14] GFan. http://apk.gfan.com/, 2015.

[15] Huawei. http://appstore.huawei.com/, 2015.

[16] Mi. http://app.mi.com/, 2015.

[17] Netease. http://m.163.com/android/, 2015.

[18] SMART: Semantic Modelling of Android Attacks. https://sites.google.com/site/droidsmat2015/, 2015.

[19] VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com, 2015.

[20] Wandoujia. http://www.wandoujia.com/apps/, 2015.

[21] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *SecureComm*, volume 127, pages 86–103, 2013.

[22] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. Machine Learning-Based Malware Detection for Android Applications: History Matters! Technical report, May 2014.

[23] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *POPL*, pages 98–109, 2005.

[24] AppBrain. Android Library Statistics. http://www.appbrain.com/stats/libraries, 2015. [Online; accessed 02-Jan-2015].

[25] 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.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, pages 259–269, 2014.

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

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

[29] D. Babić, D. Reynaud, and D. Song. Malware Analysis with Tree Automata Inference*. In *CAV*, pages 116–131, 2011.

[30] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. 41(9):866–886, 2015.

[31] J.-F. Bergeretti and B. A. Carré. Information-flow and Data-flow Analysis of While-programs. *TOPLAS*, 7(1):37–61, Jan. 1985.

[32] G. Bonfante, J.-Y. Marion, and T. D. Ta. Malware Message Classification by Dynamic Analysis. In *FPS*, pages 112–128, 2015.

[33] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[34] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based Malware Detection System for Android. In *SPSM*, pages 15–26, 2011.

[35] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *WISEC*, pages 13–24, New York, NY, USA, 2013.

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

[37] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *USENIX Security*, pages 659–674, aug. 2015.

[38] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *GbR*, 2001.

[39] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *ESORICS*, volume 7459, pages 37–54, 2012.

[40] M. De Wulf, L. Doyen, T. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *CAV*, volume 4144, pages 17–30, 2006.

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

[42] D.-F. Feng and R. Doolittle. Progressive Sequence Alignment as a Prerequisitetto Correct Phylogenetic Trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

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

[44] Gabel, Mark and Jiang, Lingxiao and Su, Zhendong. Scalable Detection of Semantic Clones. In *ICSE*, pages 321–330, 2008.

[45] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012.

[46] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *DIMVA*, pages 62–81, 2013.

[47] F. Harary and R. Norman. Some Properties of Line Digraphs. *Rendiconti del Circolo Matematico di Palermo*, 9(2):161–168, 1960.

[48] IDC. Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC . http://www.idc.com/getdoc.jsp?containerId=prUS25450615, 2015. [Online; accessed 28-Feb-2015].

[49] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[50] X. Jiang. Security Alert: New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html, 2011. [Online; accessed 13-Dec-2014].

[51] I. Keivanloo, F. Zhang, and Y. Zou. Threshold-free Code Clone Detection for a Large-scale Heterogeneous Java Repository. In *SANER*, pages 201–210, 2015.

[52] M. Labs. Threats Predictions. Technical report, 2015.

[53] E. Lafortune. ProGuard. http://proguard.sourceforge.net/, 2015.

[54] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting Differences Across Multiple Instances of Code Clones. In *ICSE*, pages 164–174, 2014.

[55] G. Meng, Y. Liu, J. Zhang, A. Pokluda, and R. Boutaba. Collaborative Security: A Survey and Taxonomy. *ACM Computing Surveys*, 2015.

[56] A. Mishne, S. Shoham, and E. Yahav. Typestate-based Semantic Code Search over Partial Programs. In *OOPSLA*, pages 997–1016, 2012.

[57] S. Oberoi, W. Song, and A. M. Youssef. AndroSAT: Security Analysis Tool for Android Applications. In *SecureWare*, 2014.

[58] N. Peiravian and X. Zhu. Machine Learning for Android Malware Detection Using Permission and API Calls. In *ICTAI*, pages 300–305, 2013.

[59] H. Peleg, S. Shoham, E. Yahav, and H. Yang. Symbolic Automata for Static Specification Mining. In *LNCS*, volume 7935, pages 63–83. Springer Berlin Heidelberg, 2013.

[60] M. D. Preda, R. Giacobazzi, A. Lakhotia, and I. Mastroeni. Abstract Symbolic Automata: Mixed Syntactic/Semantic Similarity Analysis of Executables. In *POPL*, pages 329–341, 2015.

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

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

[63] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *CASCON*, pages 13–, 1999.

[64] H. Wang, Y. Guo, Z. Ma, and X. Chen. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *ISSTA*, pages 71–82, 2015.

[65] T. Wang, S. Song, J. Sun, Y. Liu, J. Dong, X. Wang, and S. Li. More Anti-chain Based Refinement Checking. In *ICFEM*, volume 7635, pages 364–380, 2012.

[66] H. Xiao, J. Sun, Y. Liu, S.-W. Lin, and C. Sun. TzuYu: Learning Stateful Typestates. In *ASE*, pages 432–442, 2013.

[67] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In *ESORICS*, volume 8712, pages 163–182. Springer International Publishing, 2014.

[68] 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.

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

[70] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *CCS*, pages 611–622, 2013.

[71] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *CODASPY*, pages 185–196, 2013.

[72] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *CODASPY*, pages 317–326, 2012.

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

[74] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.