SAND2018-10653C

Using Loops For Malware Classification Resilient to Feature-unaware Perturbations

Aravind Machiry
UC Santa Barbara
machiry@cs.ucsb.edu

Nilo Redini
UC Santa Barbara
nredini@cs.ucsb.edu

Eric Gustafson
UC Santa Barbara
edg@cs.ucsb.edu

Yanick Fratantonio
EURECOM
yanick.fratantonio@eurecom.fr

Yung Ryn Choe
Sandia National Laboratories
yrchoe@sandia.gov

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

In the past few years, both the industry and the academic communities have developed several approaches to detect malicious Android apps. State-of-the-art research approaches achieve very high accuracy when performing malware detection on existing datasets. These approaches perform their malware classification tasks in an "offline" scenario; where malware authors cannot learn from and *adapt* their malicious apps to these systems. In real-world deployments, however, adversaries get feedback about whether their app was detected, and can react accordingly by transforming their code until they are able to influence a classification.

In this work, we propose a new approach for detecting Android malware that is designed to be resilient to feature-unaware perturbations *without* retraining. Our work builds on two key ideas. First, we consider only a subset of the codebase of a given app, both for precision and performance aspects. For this paper, our implementation focuses exclusively on the loops contained in a given app. We hypothesize, and empirically verify, that the code contained in apps' loops is enough to precisely detect malware. This provides the additional benefits of being less prone to noise and errors, and being more performant.

The second idea is to build a feature space by extracting a set of labels for each loop, and by then considering *each unique combination of these labels* as a different feature: The combinatorial nature of this feature space makes it prohibitively difficult for an attacker to influence our feature vector and avoid detection, without access to the specific model used for classification.

We assembled these techniques into a prototype, called LoopMC, which can locate loops in applications, extract features, and perform classification, without requiring source code. We used LoopMC to classify about 20,000 benign and malicious applications. While focusing on a smaller portion of the program may seem counter-intuitive, the results of these experiments are surprising: our system achieves a classification accuracy of 99.3% and 99.1% for the Malware Genome Project and VirusShare datasets, which outperforms previous approaches. We also evaluated LoopMC, along with the related work, in the context of various evasion techniques, and show that our system is more resilient to evasion.

ACM Reference Format:
Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2018. Using Loops For Malware Classification Resilient to Feature-unaware Perturbations. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3274694.3274731

1 INTRODUCTION

In the past few years, both the industry and the academic communities have developed several approaches to perform malware detection. State-of-the-art research approaches, which leverage static and dynamic analyses, are known to detect malware on existing datasets accurately. Techniques such as behavior-based signatures [46], dynamic taint tracking [21], and static data flow analysis [8, 25] are designed to locate specific symptoms or traces of *malicious behavior* (e.g., reading SMS data) in programs.

However, these approaches make the unrealistic assumption that malware authors are unaware that their code is being analyzed, or are unable to react to their code being classified as malicious. Consider, for example, Google Bouncer [32], which serves as the primary gatekeeper for Android apps published in Google's Play Store. Developers submitting apps will be notified if the system deems their app to be suspicious, allowing them to modify the app and resubmit it again. This simple model gives attackers the opportunity to learn and *adapt* their apps to hide malicious actions from a particular detection methodology. While the attacker may know some information about how Bouncer works, they do not have access to Bounder's models or detection rules, and must infer them through trial and error.

This work proposes a new technique to perform malware detection of Android apps that are resilient against feature-unaware perturbations, such as those in the above scenario. Our work builds on two key ideas. First, we consider only a subset of the codebase of a given app, both for precision and performance aspects. For

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274731

this paper, we focus exclusively on the *loops* contained in a given app. We hypothesize, and empirically verify, that the subset of a program contained in its *loops* encodes enough information to perform accurate malware detection. Loops are a key construct when writing programs, and are a key factor in the Turing-completeness of programming languages. Furthermore, loops are difficult to remove or obfuscate, and focusing on loops allows the analysis to be more robust to evasion: A functionality that *needs* loops to be implemented, *cannot* be implemented without one [13], even through techniques such as loop unrolling. While it is well understood that some loops can be transformed into equivalent structures, such as using recursion, or a high-level primitive, there is still, conceptually, a loop in the program for our analysis to find.

The second idea is to build a feature space by extracting a set of labels for each loop, and by then considering *each unique combination of these labels* as a different feature. For each loop, our analysis first determines which Android API methods are invoked, directly or indirectly. To aid our approach, we created a semantic labelling mechanism that maps every Android API to one of 202 *semantic labels*. This mechanism is used to extract, for each loop, a set of semantic labels. The *unique combination* of semantic labels is then used to create a *semantic tag*, which conceptually encodes the loop's core *functionality*. Finally, our system computes the app's feature vector by encoding *how many loops with each semantic are contained in the app*. As our approach creates a large number of features, many of them may not have any impact on classification. To mitigate this, we propose a method of iterative feature pruning, based on Random Forest feature importance, which improves performance, by reducing the number of features that need to be considered. As clearly shown in Section 4, the combinatorial nature of the considered features makes it prohibitively difficult for an attacker to influence their app's feature vector and avoid detection using feature-unaware perturbations.

Furthermore, when analyzing a previously-unseen, it may contain loops whose combination of labels is not covered by any semantic tag seen while training our model. This also provides a venue for an attacker to evade the detection. For this reason, we developed a new mapping strategy, which we call *Conservative Mapping*, that selects the nearest semantic tag, and conservatively errs toward semantic tags associated with *maliciousness*.

We assembled these techniques into a system, called LoopMC, which can be used to analyze and classify large volumes of Android applications, using only their bytecode. We evaluated our system against over 20,000 real-world benign and malicious applications both on the offline scenario, where an attacker receives no feedback, and an online scenario, where an attacker can modify the application using various techniques to evade detection. Our approach can differentiate between the malicious and benign with 99.3% and 99.1% accuracy on two different datasets. Our experiments show that our work achieves high accuracy, even when an attacker is allowed to adopt their app in response to feedback from the detection system. Even though our choice of focusing on loops may seem counter-intuitive, our experimental evaluation shows an improvement in malware detection accuracy over systems that consider the entire codebase. Finally, we compare our work with state-of-the-art malware detection systems, DroidAPIMiner and Drebin. Not only do we show that our system outperforms them, but we also show that it is easy to evade them.

In summary, this work provides the following contributions:

- We present a new approach to perform malware detection, which is resilient to feature-unaware perturbations by using machine learning models built with features extracted from loops.
- We extensively evaluate our prototype [1], LoopMC, by considering a large dataset of over 20,000 benign and malicious Android apps, and show that it can accurately identify malware.
- We show that our approach is effective even when considering an active adversary that can modify their app, while we show that existing works significantly drop in accuracy.

2 THREAT MODEL

We frame our work in the context of a typical app store or app market: developers submit apps to the store's operator to be approved for distribution, only after undergoing some series of checks and analyses. These analyses typically include those designed to prohibit the spread of malware through the store. While some details of the inner-workings of this system may be public, such as the kind of machine learning or signature mechanism in use, the exact features used, or what part of the app led to a classification, are typically kept secret from developers. Even without these details, the developer does gain an oracle against which to test future versions of their app. Therefore, we assume a threat model in which an attacker is capable of *feature-unaware perturbations*, which includes any modification that can be performed without specific knowledge of the features used in a model or ruleset used for classification. The attacker may know the method used to classify apps, such as the algorithms in use, but not the contents of the actual model. Specific to our work, this means an attacker may know that our system relies on loops, the details of our supervised machine learning technique, or the concept of semantic tags, but cannot know *which* semantic tags are considered by the trained model.

Note that this is a more realistic and stronger threat model than what has been considered by previous works, which we explore in Section 8, where the attacker is agnostic to the detection method and cannot reactively modify the app.

3 APPROACH OVERVIEW

LoopMC, whose high-level design is depicted in Figure 1, is a machine-learning-based system that uses a set of labeled APKs to train and create a model, which can then be used to detect malicious Android applications (i.e., malware). Our system works through several different stages, which are presented in this section. As previously mentioned, our work is constituted by two main phases: *loop characterization*, which performs program analysis to extract information about the loops of a given app, and *application classification*, which combines the information extracted in the first phase and uses machine learning techniques to train a model for malware detection. We provide an overview of the two main phases, each of which will be discussed in detail in the two upcoming sections.

[1] www.github.com/ucsb-seclab/LoopMC

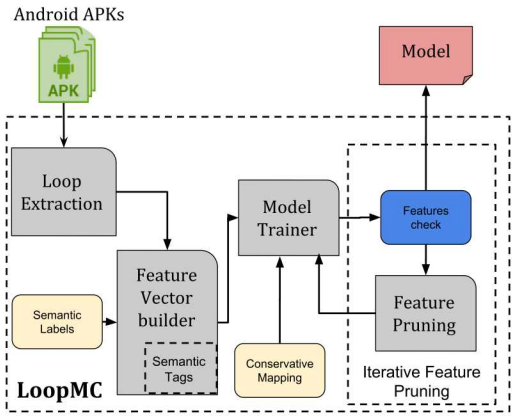


Figure 1: LoopMC overview.

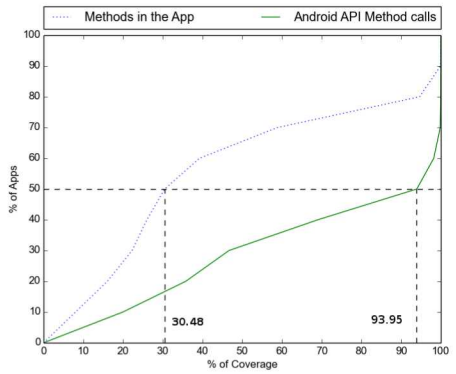


Figure 2: A CDF of the percentage of *reached* methods vs. the percentage of different Android APIs invoked by only considering each app's loops. For 50% of the apps in our dataset, loop code reaches only 30.48% of the methods, while reaching 93.95% of the different Android API call sites in the app.

Loop Characterization. This phase gets as input an Android app (i.e., an APK file), and it performs static analysis to extract all loops in the app. The analysis then considers each loop and extracts which Android framework APIs are invoked within the loop's body. Each of the invoked API is then associated with a semantics-carrying label. The unique combination of these labels forms a *semantics tag*.

Application Classification. The semantics tags extracted for each loop are used to compose a feature vector. In particular, each element of the feature vector encodes the number of loops that are associated with a specific semantics tag. Due to the combinatorial nature of the features vector, only a subset of all possible combinations are considered. In our work, we create a feature for every combination we observe during the training phase of the machine learning models. Then, we developed a mapping mechanism that allows us to take into account loops whose semantic tag has not been seen during the training phase. Our approach then applies supervised machine learning techniques to train a model to perform malware detection.

3.1 Why Loops?

Before giving details of how LoopMC works, it is important to linger on the benefits of focusing on loops. We conjecture that loops represent an essential subset of the whole program and that they can be used to infer interesting properties about the app, such as whether it is benign or malicious. Loops, in the most general sense, are made possible by a conditional branching structure and the ability to jump backward in code. Repetition is such a basic concept that most important algorithms simply cannot be expressed without it.

On the one hand, focusing on loops has two advantages. First, because of their fundamental nature, loops are more difficult to remove or obfuscate than linear code [16, 27]. This aspect makes loops good candidates to focus on. Second, as we will show in our evaluation, focusing on loops offer a significant performance boost.

On the other hand, by only focusing on loops, one may be concerned about missing important behavior of a given app. To this end, we performed a simple empirical study to understand how much relevant information is included in the loops of Android applications. To measure "relevant information," we consider two aspects: the percentage of methods invoked or *reached*, directly or indirectly (through other method calls), exclusively within apps' loops, and the percentage of different Android APIs invoked within loops with respect to the number of Android APIs invoked by the app when considering the entire codebase. (As in many related works, e.g., [3], Android APIs invoked by an app are used as a proxy to determine its behavior.)

Our results, shown in Figure 2, are surprising: for 50% of the apps in our dataset (discussed in Section 5), the apps' loops reach only 30.48% of the methods they define, but invoke up to 93.95% of the different Android APIs invoked by the app. In other words, *while loops do constitute a subset of the app's codebase, they cover most of the different Android API call sites the app contains.* It is thus not surprising that, as discussed in our evaluation, the accuracy results of our analysis do *not* improve when considering the entire codebase with respect to only considering loops.

3.2 Loop Characterization

The first major task of our system is to transform an Android app into a feature vector representing the behavior of its loops, in terms of Android's APIs. This includes recovering the actual loop structure from an app's bytecode, as well as how the loop's content is formed into the final feature vector.

Loop Extraction. Given an APK, the very first step involves its loops analysis and extraction. Particularly, LoopMC disassembles the app's Dalvik bytecode and compute the APK's static control flow graph (CFG), which is then used to locate the application's loops. In our context, a "loop" is not just an instance of a looping construct available in the programming languages (for, while, do, etc.), but any cycle in the CFG. We leverage a well-known Depth-First-Search-based (DFS) loop-finding algorithm [43] that reliably detects all loops in the program. In our implementation we relied on Androguard [17] to perform the above-mentioned steps.

Semantic Labels. In this work, we characterize a loop's effect on the state of the program by determining which Android framework APIs could be possibly invoked from its guard or body. This is accomplished by considering the *transitive closure* of all guard and body code, and consider all API method calls it contains. To this end, we scan the bytecode for `invoke` instructions. In order to resolve

the actual function being called, we must determine the type of the object whose method is being called, which may have more than one solution. This effectively simulates *dynamic dispatch*, the mechanism that, at run-time, determines which method to call according to the dynamic type of the receiving object. To do this, we perform a Class Hierarchy Analysis (CHA) to determine the complete set of possible classes and the contained methods that could possibly be invoked by the considered `invoke` instruction.

The number of possible Android API calls is very large, about 65,000, and determining the semantics of each of them is a challenging, open problem. What makes this aspect difficult is that some APIs might perform functions unrelated to their name, class, or package hierarchy. On the other hand, two APIs with similar names might perform very different tasks. To solve this, we assign to each Android framework API a *semantic label*. In particular, we defined a set of 202 semantic labels, which assign a semantics to the net effects of each API invocation. For example, the method `saveAttributes()` of `android.media.ExifInterface` is assigned the label `ioWrite` as it is used to write EXIF tags to a JPEG file, which in effect is a file writing operation. As a starting point, we considered results generated by automatic tools proposed by previous research [7, 9]. We then carefully examined the Java and Android documentation to assign labels to every API method that we encountered in our dataset. Fortunately, not every individual API needs to be labeled separately, and we can map several classes and methods to a label through simple pattern matching.

Although this process was mostly manual and tedious, it is a one-time effort and is needed to address several imprecisions that necessarily occur when using automatic tools. We released [2] the "Semantic Labelling Mapping" code and data, accompanied by the information on how to extend it. We aim to make this a community-driven effort, which could be useful even outside the scope of our work.

Semantic Tags. As the final step to characterize each loop, we consider the *unique combination of its semantic labels* to create what we call a *semantic tag*. In particular, we only consider whether a specific semantic label is used or not. Intuitively, the number of times a certain label appears is not significant, since the code in a loop is meant to be repeated. Note that, a semantic tag represents a dimension in the feature space on which our *Model Trainer* works. In other words, a semantic tag is a feature in the APK's feature vector.

3.3 Application Classification

The second phase of the system involves using the set of semantic tags for an app to classify it as malicious or benign. Each element of the feature vector encodes the number of loops that are associated to a specific semantic tag. In our work, we first create a feature for each semantic tag we observed during the training phase of our model. We may not have seen all the possible semantic tags during training, to handle this we developed a mapping mechanism that allows us to take into account the loops whose semantic tag has not been seen during the training phase. Our approach then applies supervised machine learning techniques to perform malware detection.

[2] www.github.com/ucsb-seclab/LoopMC

Feature Vector Builder. To classify entire apps, we need a way of capturing what each app as a whole *does*, using the features we collected. For each app, we record the number of different loops sharing the same semantic tags. In this context, we do take into account the repetitions of the same feature, since they represent additional code within the app.

As an example, consider an app with five different loops with the following semantic tags: two loops with the semantic tag `{database, string, datastructure}` and one each of `{iterator, genericFileOp, datastructure}`, `{database, UI}`, and `{networkRead, string, UI}`. Assuming that these four are the only semantic tag in our dataset, the app's final vector would be `{2,1,1,1}`. Of course, in a real scenario, we would have many more features, and so the vector would be very long and sparse (i.e., many feature values would have value zero).

Model Trainer. We have a large feature space because of the combinatorial nature of semantic tags, and this makes the feature selection infeasible. However, feature selection is a required step to apply well-known classifications algorithms based on k-nearest neighbors, support vector machines and neural networks. Furthermore, these algorithms do not provide insight into the importance of each feature, which we use in our conservative mapping to handle unknown features [5].

Therefore, a decision tree-based system was a natural choice, as it does not depend on the meaning of distance in the feature space (in our feature space the concept of distance is meaningless), and provides importance measures for each of the features [5]. Consequently, we applied the Random Forest algorithm to the extracted features to create a model for detection.

Note that, though in principle decision-tree-based algorithms suffer from the *curse of dimensionality* [28] due to a big feature space (like ours), Random Forest tackles this problem [37] by employing an ensemble of decision trees as well as the bootstrapping scheme. In fact, as clearly shown in Section 5, even when the feature space is drastically reduced from dozens of thousands of features to a few dozens, the decrease in accuracy is either irrelevant or non-existent.

Also note that our system can also handle feature vectors that do not align with those used to build the model, as described later in this very section. For details about how Random Forest models are trained and evaluated, we direct the reader to the exhaustive documentation on the topic [10–12, 33].

Random Forest has few parameters to tune, and of these, only the number of trees had any meaningful impact on our experiments. We explored the optimal number of trees to use in the ensemble and found that after 50 trees, accuracy changes became insignificant, oscillating within 0.05%. All model-building is done under 10-fold cross-validation, which was determined to be optimal for real-world datasets in [29]. Accuracy is reported as the maximum accuracy of the ten models.

Iterative Feature Pruning. We implemented an approach to feature pruning that provides for a degree of scalability, enhanced accuracy, and gives new insights into the data. While the feature-space of our data is somewhat large, many of these features do not have any effect on whether the sample is malicious or benign, in any possible context.

We propose a method of iterative pruning that removes unimportant features from the model, without using arbitrary thresholds. First, we train a model, using all of the features in our dataset. Then, we calculate the *importance* [24] of each feature, and remove those that we know have no effect on classification (i.e., have an importance of zero across all trees). Finally, we retrain a new model, only considering the remaining features, and repeat the process until no additional features can be removed. In simple words, we can say that a feature is more important if it produces a better split between the two classes. This approach is similar to the non-iterative one proposed by Appel et al. [4], although as we will show in Section 5, the iteration allows us to remove even more under-performing features.

Conservative Mapping. When built with a large-enough initial dataset, our model should ideally contain the vast majority of possible semantic tags that will be seen in Android applications. However, we must account for the cases where a combination of API calls results in a semantic tag that is not part of the feature vectors. This can happen simply in the natural course of writing programs, but also due to a deliberate attempt to evade detection. In the default case, where only semantic tags in the original model are considered, this could lead to a decrease in accuracy over time.

To mitigate the effects of this potential attack strategy, our system supports a second mode of classification, which minimizes these issues, at a very slight accuracy penalty. In this mode, we construct a model using only the features that are considered important to the classification of a malicious sample. First, we build a model, using the standard procedure described previously in this very section. From this model, we then determine the *Malware Importance*, or MI, of each feature. Finally, we construct a new model, using only those features whose MI is greater than zero. To classify unknown samples, we map new features to those in the model, with a bias toward mapping to those with a higher MI.

The above process hinges on the computation of the *Malware Importance*, which is a non-negative real number indicating the importance of a feature in deciding that a sample is malicious. This is built on top of the traditional concept of feature importance in Random Forests [24].

For each non-leaf node (cn) in a decision tree of the model, which contains a feature test of the form $value(semantic\_tag) > threshold\_value$ (where $value(semantic\_tag)$ is the value of the feature represented by $semantic\_tag$), the MI is computed as follows:

$$MI(semantic\_tag) = \begin{cases} \frac{MS(tc)-MS(fc)}{N_{root}} & if\ MS(tc)>MS(fc) \\ 0 & otherwise \end{cases}$$

where N_{root} is the number of samples (both malware and benign) at the root node of the decision tree, tc and fc are the sub-trees traversed when the feature test is true or false respectively and MS indicates the number of malware samples that reached the node in the corresponding decision tree during training. If $MS(tc) > MS(fc)$, more malware samples have values of the feature (or semantic tag) above the corresponding $threshold\_value$, which captures the notion of local importance. Dividing by N_{root} brings in the effect of the position of the feature in the decision tree, thus capturing a feature's global importance.

Semantic tag or Features	MI
<iterator >	0
<sms, iterator >	0.32
<audio >	0
<androidCursor >	0
<reflection, ioWrite, networkRead >	0.013
<ioWrite, networkRead >	0.004
<ioWrite, ioRead >	0.00016
<reflection, iterator, networkRead >	0.023
<ioRead >	0

Table 2: Example of set of semantic tags with corresponding MI

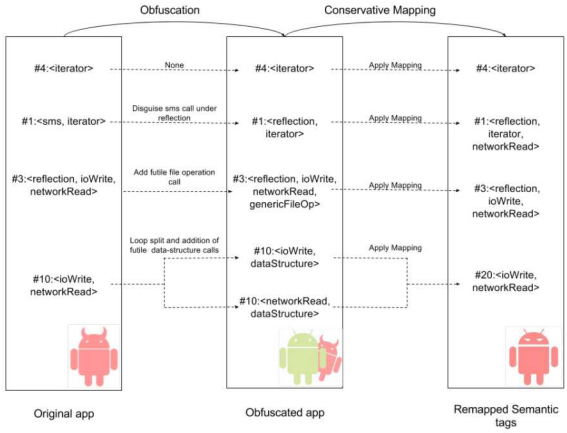


Figure 3: Conservative Mapping example

If a feature is used in more than one tree, we pick the maximum value of the MI across all trees in the forest.

With the importances calculated, we can then perform the Conservative Mapping step. For a given semantic tag (i.e., *src*), we find the closest semantic tag (i.e., *dst*) with the least symmetric difference in their labels and a non-zero MI. We resolve ties by picking the tag with the greater MI.

Here we will present a realistic example of the Conservative Mapping in action. Table 2 shows the entire set of semantic tags found in the hypothetical dataset of apps used for training. Figure 3 depicts what would happen when an app with loops of these tags are obfuscated through various strategies, as well as how our mapping handles unknown tags this process creates. The first box of this figure represents a malware sample whose loops are represented in the form $\#number\_of\_loops :< semantic\_tag >$. The feature vector corresponding to features in Table 2 is: <4,1,0,0,3,10,0,0,0>. As shown in this figure, the attacker adds useless API calls in the body of two existing loops disguises an SMS-related API call with the use of reflection, and finally splits a loop into two different loops, adding a useless API call in both of them. The resulting app is shown as the second box in Figure 3. Using the default mapping, where unseen semantic tags are ignored, the feature vector would be <4,0,0,0,0,0,0,0,0>, resulting in benign classification. However, by enabling Conservative Mapping, the obfuscated loops would map to a different set of semantic tags based on the Malware Importance. The resulting feature vector is: <4,0,0,0,3,20,0,1,0>, consequently classifying the app as malicious. Table 1 shows the reasons the semantic tags represented in the third box of Figure 3 were chosen

API label	Best semantic tag matched	Reason
<iterator >	<iterator >	Exact Match
<reflection, iterator >	<reflection, iterator, networkRead >	Closest tag with a higher MI
<reflection, ioWrite, networkRead, genericFileOp >	<reflection, ioWrite, networkRead >	Closest tag
<ioWrite, dataStructure >	<ioWrite, networkRead >	Closest tag with a higher MI
<networkRead, dataStructure >	<ioWrite, networkRead >	Closest tag

Table 1: Tag matching procedure

from the semantic tags provided by this particular system (Table 2). The conservative mapping, as we will show in Section 5 may raise the false-positive rate, but also allows our system to easily withstand code obfuscation attacks.

4 RESILIENCE TO FEATURE-UNAWARE PERTURBATIONS

This section discusses the resiliency of LoopMC against various evasion attempts. Since the features of the app have been componentized into their loops, merely spraying API calls into a method does not skew the app's features, as it does with the related work. However, there are other evasion techniques that an adversary could pursue if they were aware of our system.

4.1 Application Transformations

The transformations, while simple to implement and perform, have a dramatic impact on the detection rates of off-the-shelf anti-malware solutions. Table 3 shows a list of the transformation techniques available in DroidChameleon [38], their effects on the program, and their effect on the classifications performed by LoopMC. First, any transformation that merely alters the data or human-readable strings of the app, such as string encryption or class renaming, has no effect whatsoever on our system's feature vectors. Second, LoopMC considers the transitive closure of a loop's code, and Call Indirection and in/outlining will not affect it. Our system would be affected by bytecode encryption, as the original bytecode is simply unavailable to static analysis. However, the mere decrypting, loading, and executing bytecode can be used as a strong signal to detect Android malware [20, 26, 36]. We note that this may not be true for classic desktop malware [2].

4.2 CFG Obfuscation

While LoopMC is resilient to many program transformations, one of the more interesting aspects of its performance involves mutation of the program's control-flow graph (CFG). CFG obfuscation works by adding unfeasible random forward and backward jumps, which may appear as loops. This represents a worst-case scenario for our system, as similar work that leverages program-wide sets of API calls are unaffected. Changing the CFG can impact how loops are extracted, and change our feature vectors. However, as we will show in Section 5, we are still able to detect malicious apps correctly.

4.3 Reflection

One way in which an attacker can evade an API based malware analysis systems is through the use of Reflection to hide method calls. This could happen by converting some or all API calls to the equivalent expression using `java.lang.reflect.Method` and similar. This would, at a minimum, allow an attacker to alter the features extracted by any system that uses API calls, including ours.

In a simplistic scenario, an attacker could use one of the many tools [14] available for Java programs to automate the obfuscation and transform all method calls into reflection. These tools provide an easy way to evade all API call-based detection systems. Complex string analysis is required to resolve the invoke targets, and maintain accuracy. Secondly, a clever attacker can convert some API calls which they believe to be the most malicious into reflection. For example, the adversary may choose to obfuscate only methods dealing with sending and receiving SMS, a popular feature of Android malware [47]. We explore both scenarios experimentally in Section 5.

4.4 Loop Perturbations

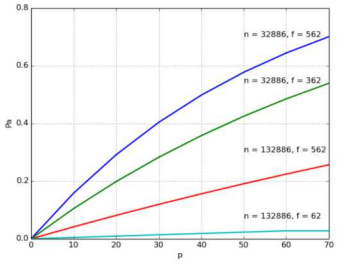


Figure 4: Relation among n, p_a, f and p.

In a real-world deployment of our system, the attacker may perform *Loop Perturbations*, which we define as the ability to modify their program, by knowing the LoopMC algorithm, but without the detailed knowledge of the model being used (i.e., the employed feature vector is unknown). We further define these to be passive perturbations (e.g., those that do not affect the app's overall behavior). Specifically, we consider spurious code insertion [35], as it is much simpler to generate meaningless code without concern for the impact on the program's execution. We will evaluate the difficulty for an attacker to evade our system. As a difficulty *metric*, we can compute the probability that randomly-inserted code with API calls in loops can influence the feature vector.

Let's consider n to be all possible features available to the model, and f to be the number of important features selected by the underlying detection system to classify the app. For a given n, we estimate the number of features (p) to be chosen from n, so that at least one of the features will be in f with some probability (P_a). The

Transformation Name	Description	LoopMC is Resilient?	Comments
Changing Package Name	Package name is changed in AndroidManifest	YES	LoopMC does not use the Manifest file
Identifier Renaming	field names (both local and static) are renamed	YES	LoopMC does not rely on the names of any fields.
Data Encryption	Strings and Data used in the code are encrypted and decrypted in place.	YES	LoopMC is data-agnostic.
Call Indirections	Add wrapper functions to API calls	YES	Transitive closure ignores indirection.
Code Reordering	Reordering the instructions and inserting goto instructions to preserve the runtime execution sequence of the instructions	YES	Content of loops is preserved. Algorithm [43] for loop detection is resilient to this.
Junk Code Insertion	Insert junk code which does not affect the semantics	YES	LoopMC only considers calls to API methods
Function Outlining and Inlining	Split a function into multiple functions and Replace a function call with the entire function body	YES	Transitive closure adds resilience to this.
Encrypting Payloads and Native Exploits	Code is stored as encrypted blob, decrypted at run time and executed.	NO	Our analysis is static and fails to identify the execution of decrypted code.
Bytecode encryption	Relevant piece of the application code is stored in an encrypted form and is decrypted at runtime via a decryption routine	NO	Presence of decryption routine at the start of an application could be used as an indication of malware.

Table 3: Evaluation of LoopMC against various Application Transformations

relation between, p,n,f and P_a is given by the following equation:

$$P_a = \begin{cases} 1 - \dfrac{\binom{n-f}{p}}{\binom{n}{p}} & \text{if } (n\text{-}f) > p \\ 1 & \text{otherwise} \end{cases}$$

The first case of the above equation represents the probability of affecting at least one feature. The second case handled the scenario when we chose more than $(n-f)$ features, where we can guarantee that at least one feature will be in f.
Here, n represents the choices available for the attacker, f ($\subseteq n$) is the secret set of items that they have to guess, p represents the number of items to be chosen from n where they could be reasonably sure ($P_a > 0$) that one of the items belongs to f. From the above equation, the following observations could be made:

(1) For a fixed n and P_a, increasing f decreases p (i.e., $f \propto \frac{1}{p}$).
(2) For a fixed f and P_a, increasing n increases p (i.e., $n \propto p$).
(3) For a fixed n and f, increasing p increases P_a (i.e., $p \propto P_a$).
(4) For a fixed n and p, increasing f increases P_a (i.e., $f \propto P_a$).

Figure 4 illustrates these observations with some sample values for n and f. In Section 5.5, we show how our features make it relatively hard to evade detection by our system using blind perturbations.

Note that in the above equation, we assume a uniform distribution of all the features. However, in practice, the features may be biased based on common characteristics of Android malware, which could affect P_a. For instance, the probability of having `{iterator, sms}` in the model is more than `{iterator, uiObjectRead}`. However, as we show in Section 5.5, our n is very large, which minimizes this effect.

5 CLASSIFICATION EVALUATION

In this section, we evaluate LoopMC's effectiveness at classifying malicious apps only based on information in their loops. First, we will explore the performance of LoopMC at classification, including an isolated evaluation of Iterative Feature Pruning and Semantic Labeling. We will then show the resilience of our system, compared to previous work, in a scenario where an attacker can obfuscate the CFG of applications.

As we discuss in Section 8, our work most closely relates to the DroidAPIMiner [3] and Drebin [6] systems. Both systems leverage the total set of API calls and permissions of the applications to compose their feature vectors, and use machine learning to classify applications.

While we were unable to obtain the actual source code for either system, we were able to reproduce DroidAPIMiner's approach. Using the same static analysis framework that LoopMC is built on, we extract those features that appear more frequently in malicious apps by a 6% margin (as in [3]). We then classify the applications using RapidMiner's k-nearest Neighbor implementation (k=1), under 10-fold cross-validation. This allows us to compare our system to DroidAPIMiner under obfuscation scenarios.

5.1 Datasets

We evaluated LoopMC with two different publicly-available malicious app datasets. The first, the Malware Genome Project dataset [47], contains 1,260 apps (178,795 loops) from various malware families. While this dataset is small, and the repetition of samples from different families does not provide a lot of variation in the included apps, this is the best available benchmark to compare against related work. To get a better understanding of how the system functioned in more realistic circumstances, we obtained 11,080 malicious apps from the VirusShare project [1]. From these,

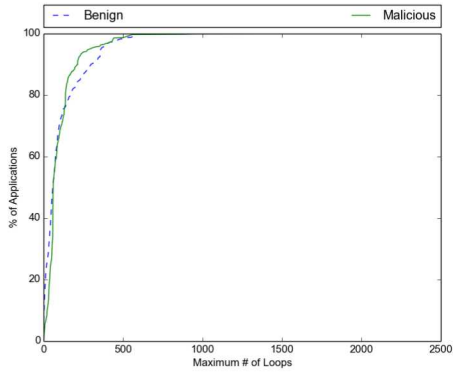


Figure 5: CDF of loops present in malicious and benign datasets

we used 6,016 apps (653,855 loops) in our experiments, as Androguard failed to disassemble the rest [19]. These apps are of unknown distribution but likely vary widely in source and malware family. As we will show in the following sections, our results support our assumptions regarding the variety in the dataset. For benign data, we obtained 20,000 presumed-benign applications crawled from the Google Play store using the PlayDrone tool [42], from which we ignored 931 apps as they were very simple example apps with less than 5 loops. We are able to successfully analyze 17,414 apps (8,965,146 loops) and 1,655 timed out (took more than 20 minutes to compute the transitive closure of API invocations) because of the issues with the version of Androguard [18] we are using.

Initially, we suspected there were inherent differences in the size or code complexity of the two classes. We anticipated that malware would be smaller, due to only needing to perform the intended malicious functionality. One way we can measure this in our data is to examine the distribution of the loops in both classes. Figure 5 shows a CDF of the number of loops in the apps from both classes, using the larger VirusShare dataset. From this, we can deduce that there is no substantial difference in the number of loops and classifying this data is not so simple, and it will serve as an excellent test of our system.

5.2 Iterative Pruning Performance

Here we will demonstrate the performance of the iterative feature pruning technique used in our system, as introduced in Section 3.3.

Figure 6 shows the performance while analyzing our dataset, in terms of the change in the number of features during each iteration, as well as the accuracy of the model after 10-fold cross-validation. The process converged in few iterations (nine for VirusShare and six for Genome respectively), had no negative impact on the cross validation or accuracy scores, and produced a massive decrease in the number of features (from 124,781 to 440 and from 47,881 to 38 in the VirusShares and Genome datasets respectively).

5.3 Malware Classification Results

Table 4 shows the results for our system, when analyzing unobfuscated samples, and compares them with existing malware detection systems. The results for DroidAPIMiner and Drebin with

System	Accuracy		System Type	Evaluation against Evasion
	Malware Genome	Virus Share		
LoopMC	99.3%	99.1%	Off-device	Yes
DroidAPI Miner	99%	97.4%	Off-Device	No
Drebin	94%	–	On-Device	No

Table 4: Comparison of LoopMC against DroidAPIMiner [3] and Drebin [6].

the Malware Genome dataset were obtained from their corresponding papers, and the results for VirusShare were determined from our re-implementation of the DroidAPIMiner system. Drebin was unavailable to run on the VirusShare dataset. In the above tests, our system presents the very low False Positive rate of 0.5% and 0.3% for the Malware Genome and VirusShare datasets, respectively.

5.4 Importance of Loops and Semantic Labels

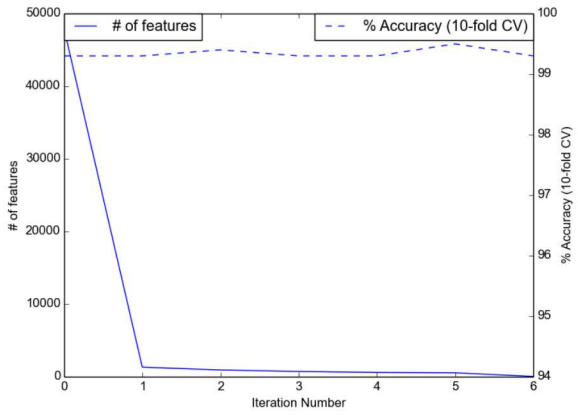
As one might argue that the effectiveness of our system is mainly due to a careful and precise choice of semantic labels, in this section we dissect our system into its two main components, and we show that they both contributed to its detection effectiveness.

First, we show that, independently of the malware detection mechanism being used, loops alone encode enough information to allow it to precisely detect malicious applications, as well as greatly speeding up the overall detection process. To prove this, we ran our DroidAPIMiner implementation on the same Malware Genome dataset described above but considering only the code that is reachable within the loops of the apps. This resulted in a classification accuracy of 96.15%, only a slight decrease from when the entire program is considered. This result clearly shows that independent of the detection technique, API behavior captured by the loops can be used to detect malware. Furthermore, since loops are indeed a small portion of the app's code, this represents a significant optimization, without a significant sacrifice in accuracy. In fact, we explored a version of our analysis that focuses on methods, instead of the loops as the basis for features. While the accuracy was similar (within 1%), the analysis time per app increased by at least 10x, as there are many more methods to analyze than there are loops. For instance, in the Malware Genome dataset, there are 1,723,413 methods, compared to the 178,795 loops that our system extracted.

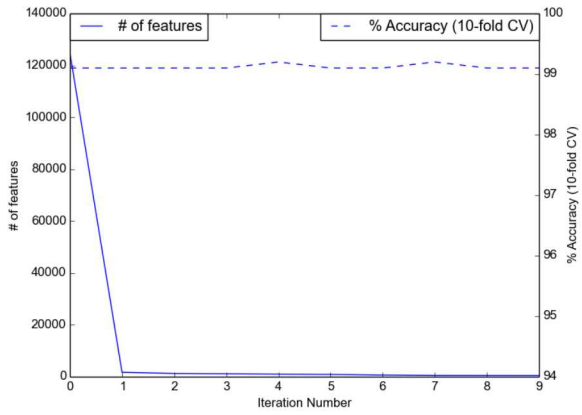
The semantic labels used to create LoopMC's features are, in part, derived from manual inspection of APIs. To understand the impact this has on our approach, as well as that of previous work, we evaluated this aspect independently. We performed the same DroidAPIMiner experiment outlined above on the Malware Genome dataset but substituted the API calls themselves for their semantic label. This resulted in 52.37% accuracy. This shows that manually created semantic labels alone are not effective and LoopMC gains its advantage from its ensemble of techniques and not just from the semantic labels.

5.5 Resilience to Feature-unaware Perturbations

In this section, we experimentally evaluate the resilience of our system to evasion attempts, compared to previous work, including



(a) Genome dataset



(b) VirusShare dataset

Figure 6: Effects of iterative pruning on the number of features and model validation under cross-validation.

Dataset	True Positive	False Positive
Malware Genome	99.99%	2.01%
VirusShare	99.79%	5.45%

Table 5: Performance on CFG-obfuscated malware with Conservative Mapping

System	Configuration	Precision
	Max Increase of feature Value	
LoopMC	10	99.10%
	15	97.03%
	20	96.25%
	25	95.60%
DroidAPIMiner	Random bits set	92.51%

Table 6: Additive obfuscation performance of DroidAPIMiner and LoopMC

permutations of the CFG and hiding malicious behaviors via reflection. We have discussed other forms of program transformations in Section 4.

CFG Obfuscation. We evaluated our system with ADAM, a framework designed to test malware detection approaches [45]. ADAM contains different modes, including injection of static code, name obfuscation, string encryption, and CFG manipulation. Here we will focus on CFG Obfuscation, which involves the injection of spurious back-edges in the CFG; the other modules do not affect our system, as outlined in the previous section.

Table 5 shows the results when running LoopMC on samples from the Malware Genome and VirusShare datasets, obfuscated by ADAM. Even in the worst case, when employing the Conservative Mapping, false positive increase by only 1.51 and 5.15 percent respectively.

Reflection. Here we will evaluate the effects of an attacker replacing the desired malicious behavior with the equivalent method calls using reflection. First, we explore the scenario in which every API call in the app has been converted to a reflection-based call. We simulated this scenario by replacing all API calls with the reflection methods and generating the corresponding feature vectors. *DroidAPIMiner classified all these malware samples as benign. In contrast, LoopMC with Conservative Mapping classified all malware correctly.* We note that both approaches lose visibility in what the actual behavior of a given app is (i.e., all method calls appear to be *reflective* calls), but the conservative nature of our approach can minimize false negatives. However, when reflection is applied to all the methods in benign apps, LoopMC classified them as malware too demonstrating that Conservative Mapping could be aggressive and results in false positives in extreme cases.

Second, an attacker could only use reflection to invoke certain functions of interest. We simulated obfuscation of all the malware samples in the Malware Genome and VirusShare datasets, by replacing all the SMS related API calls with reflection invocations. *On the VirusShare Dataset, Accuracy of DroidAPIMiner dropped to 83.54% (with a 13.86% decrease), while LoopMC's accuracy is relatively high at 92.47% (with only 4.63% decrease).* However, *on the Malware Genome Dataset, the accuracy of both systems remained unaffected.* After careful inspection of the malware samples, we noticed that these samples were glaringly malicious with several samples doing multiple malicious activities [47]. This explains why obfuscating SMS related API calls alone has no effect on their classification, whereas the VirusShare dataset represents a larger, more modern, and complete dataset.

Blind Perturbations. As we explain in Section 4, p represents the attacker effort required to change the feature vector with probability P_a, given that the attacker has no access to the internals of the model used for classification, or the dataset used to train it. In API-based detection systems, such as DroidAPIMiner and Drebin, n is the total number of API methods, f is the API methods which are frequently seen in malware, and used as features. For the Malware Genome Dataset, $f^{DAM}_{genome} = 735$, whereas $n^{DAM} \approx 65,000$. Consider p^{DAM}_{genome} to be the number of API methods that need to be added to affect DroidAPIMiner feature vector with probability $P_{reasonable}$.

In LoopMC, as mentioned in Section 3, we have 202 possible labels, and a semantic tag is a subset of these labels. Hence, the total number of possible semantic tags, n, which is equal to the power set of all labels, is equal to $n^{\text{LoopMC}} = 2^{202}$, and the number of features used after iterative pruning is equal to $f^{LoopMC}_{genome} = 38$. Consider $p^{\text{LoopMC}}_{genome}$ to be the number of semantic

tags (i.e., loops) that needs to be added to affect the LoopMC feature vector with probability $P_{reasonable}$. We have $n^{\text{LoopMC}} \gg n^{DAM}$ and $f^{\text{LoopMC}}_{genome} < f^{DAM}_{genome}$. From observations 1, 2, 3 and 4, since P_a is fixed, we have $p^{\text{LoopMC}}_{genome} \gg p^{DAM}_{genome}$.

Moreover, as we use a decision tree-based classifier, even if important semantic tags are known, it is not trivial to affect classification. An attacker must add features that allow a sample to escape all the decision nodes in the model by affecting the corresponding features. In contrast, it is comparatively easy to affect the classification of DroidAPIMiner, or any other system that samples these features from the entire app. The attacker can know important framework methods (f) from the most used framework methods in malware and can affect the classification of malware by adding those framework methods.

We also show, using a simulation that even if a change is performed, it has little to no chance of affecting classification. More precisely, we simulate the effect of adding code, by altering the feature vectors from the Malware Genome dataset used by both LoopMC and DroidAPIMiner. For DroidAPIMiner, this means changing random bits in the feature vector from 0 to 1. For LoopMC, we randomly increase some features in the vector. We use LoopMC's default classification approach for this evaluation.

Table 6 shows the results of our simulation. Even though random features (representing random code) were added to LoopMC's feature vectors, this did not affect its precision. In contrast, DroidAPIMiner's precision dropped from 99% to 92.51%. We can conclude from these experiments that, not only it is extremely difficult to influence our feature vectors, but also the result of doing so does not affect our precision as much as the related work.

6 DISCUSSION

Our current prototype does not support the analysis of loops that are implemented through recursion. There is no fundamental limitation that prevents the proper modelling of these aspects into our framework, and we plan to extend our framework as part of our future work.

There are other well-known transformation techniques that affect the loops and could potentially influence the detection from our system:

- **Loop unrolling**: This is a transformation technique where, if the number of iterations of a loop is known, then the body of the loop is repeated the corresponding number of times thus eliminating the loop. However, as shown by a recent large-scale study [23], android applications contain very few loops whose number of iterations can be determined statically.
- **Inserting junk or infeasible loops**: As we do not evaluate the feasibility of a loop execution, inserting junk loops (e.g., unsatisfiable guard conditions, or unreachable loops) could affect the feature vector and hence our detection. ADAM inserts infeasible loops into any given Android app but, as we show in Section 5.5, our system was able to detect malware even when they were obfuscated by ADAM.
- **Merging and Splitting the loops**: Merging and splitting the loops could change the semantic tag of the original loop and potentially could result in unknown semantic tags. However, as we show in Section 3.3, we handle this by using our conservative mapping approach. Furthermore, loop splitting [31] is a known hard problem that requires precise resolution of data-dependencies and pointer aliases, which makes this particular attack vector hard to use even for experienced attackers.

We acknowledge that existing works like DroidAPIMiner and Drebin might be improved to be resilient to perturbations. However, they still could be easily affected as they depend on the entire codebase of the app unlike LoopMC which leverages only loops.

7 LIMITATIONS

We believe our work represents a significant step forward in the detection of mobile malware. However, we acknowledge that our approach is affected by the following limitations.

- **Static analysis evasion**: Our system cannot handle those apps using techniques such as bytecode encryption, packing, malicious native code, dynamic code loading, or VM-based obfuscation that actively try to evade static analysis. We note that these techniques also affect all the other works that depend on static analysis. Moreover, in certain contexts, such techniques may appear suspicious enough to lead to a malicious classification.
- **Interactions with Android framework**: Android is an event-driven system that allows apps to register for call-backs on the occurrence of certain events. In the current implementation, we do not model the call-back functions, which may result in missing edges in the call-graph. However, Cao et al. [15] proposed a technique which could be used to handle the framework interactions and achieve a complete call-graph.
- **No Loops Malware**: As LoopMC depends on the presence of loops, a simple malware that has no loops can easily evade our system. However, we can have a filtering phase where apps without loops could be filtered and alerted for alternative detection techniques.
- **Dependent tools**: Our analysis system is implemented by leveraging Androguard, consequently LoopMC inherits the issues and limitations of Androguard as demonstrated by the apps which timed out during our analysis.

8 RELATED WORK

This section compares our work against recent advances in malware detection and Android program analysis. The most similar works are those that focus on machine-learning-based approaches to detecting malicious Android programs. Aafer et al. propose DroidAPIMiner [3], which uses the set of Android API that a given application can invoke to build a features vector. The authors then explore several machine learning algorithms, and they empirically established that the k-nearest Neighbors technique outperforms the other ones. Arp et al. propose Drebin [6], another malware classification system, which uses an SVM-based technique that takes into account the app's required permissions when building the feature vectors. A different system is AppContext [44], which uses machine learning techniques to identify malware by using the "context" of

each behavior as a feature. Finally, Garcia et al. recently published a technical report that describes RevealDroid, a malware detection and classification system that combines flow-related information provided by FlowDroid [8] to information related to sensitive API call invocation, security-relevant data flows, and Intent-related actions.

In contrast, our approach only needs to analyze the subset of a program contained in its loops. Therefore, LoopMC only needs to analyze a reduced (yet meaningful) code base, and, by doing this, the approach can scale better and is more robust to evasion, since the mere insertion of API calls is not sufficient to modify the classification outcome.

Recently researchers developed MAMADROID [34], which uses `java` package names instead of API function calls to build behavioral models. However, as we show in Section 3, this is not always correct, as methods of different packages could have the same semantics. LoopMC uses a semantic labeling scheme that is a one-time effort to provide a precise and scalable labeling technique. Other approaches leverage static analyses to locate malicious behavior without executing the app. Kirin [22] uses a rule-based scheme to detect dangerous configurations of permissions. FlowDroid [8] and DroidSafe [25] propose static taint analyses to detect potentially malicious data flows. Additionally, RiskRanker [26] and DroidRanger [48] rely on symbolic execution and heuristics to identify and rank malicious behavior. Although these works share our same goal, LoopMC is based on a very different technique, and it achieves a higher classification precision.

In contrast with all of the above approaches, there has been extensive research on finding malware through dynamic analysis [21, 30, 32, 39–41]. These works are complementary to ours and all share the inherent limitations of the dynamic analysis, namely, that they are limited by their ability to stimulate the app such that the malicious behavior is exposed.

9 CONCLUSIONS

This paper explores a new approach for classifying Android malware that is resilient against feature-unaware perturbations. Our approach works by focusing on the loops of a program and by mapping each app to a very large feature space that makes it challenging for an attacker to easily change the classification outcome. We assembled these ideas into a proof-of-concept system, LoopMC, and we evaluated it with 20,000 malicious and benign Android applications. LoopMC classifies them correctly with 99.3% and 99.1% accuracy on two different datasets. We then showed, through simulations and experiments, that LoopMC is more resilient to various types of evasion techniques, including modifying the CFG, using reflection, and performing targeted feature manipulation.

ACKNOWLEDGMENTS

We would like to thank our shepherd Juan Tapiador and other anonymous reviewers for their valuable comments and input to improve our paper. This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0084 and HR001118C0060, by the Office of Naval Research under N00014-17-1-2897, and by the National Science Foundation under CNS-1408632. This research is also supported by a Google Security, Privacy, and Anti-Abuse Award. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] VirusShare.com - Because Sharing is Caring, 2015.
[2] LastLine Blog: When Malware is Packing Heat, 2017.
[3] Aafer, Y., Du, W., and Yin, H. Droidapiminer: Mining api-level features for robust malware detection in android. In *Proceedings of the Security and Privacy in Communication Networks (SecureComm)* (2013).
[4] Appel, R., Fuchs, T., Dollár, P., and Perona, P. Quickly boosting decision trees-pruning underachieving features early. In *Proceedings of the JMLR Workshop and Conference (JMLR)* (2013).
[5] Archer, K. J., and Kimes, R. V. Empirical characterization of random forest variable importance measures. *Computational Statistics and Data Analysis 52*, 4 (2008), 2249 – 2260.
[6] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., and Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2014).
[7] Arzt, S., Rasthofer, S., and Bodden, E. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114* (2013).
[8] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2014).
[9] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: analyzing the android permission specification. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012).
[10] Breiman, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.
[11] Breiman, L. Manual on setting up, using, and understanding random forests v3. 1. *Statistics Department University of California Berkeley, CA, USA* (2002).
[12] Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. *Classification and regression trees*. CRC press, 1984.
[13] Brooks, R. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies 18*, 6 (1983), 543–554.
[14] Buzatu, F. Methods for obfuscating java programs. *Journal of Mobile, Embedded and Distributed Systems 4*, 1 (2012), 25–30.
[15] Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2015).
[16] Chan, J.-T., and Yang, W. Advanced obfuscation techniques for java bytecode. *Journal of Systems and Software 71*, 1 (2004), 1–10.
[17] Desnos, A. Androguard: Reverse engineering, malware and goodware analysis of Android applications... and more (ninja!). https://code.google.com/p/androguard/.
[18] Desnos, A. Androguard: Reverse engineering, malware and goodware analysis of Android applications... and more (ninja!). https://github.com/androguard/androguard/issues.
[19] Desnos, A. Bugs in AndroGuard to disassemble APKs. https://github.com/androguard/androguard/issues.
[20] Duan, Y., Zhang, M., Bhaskar, A. V., Yin, H., Pan, X., Li, T., Wang, X., and Wang, X. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2018).
[21] Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., and Sheth, A. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2010).
[22] Enck, W., Ongtang, M., and McDaniel, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the ACM Conference on Computer and*

Communications Security (CCS) (2009).

[23] Fratantonio, Y., Machiry, A., Bianchi, A., Kruegel, C., and Vigna, G. Clapp: characterizing loops in android applications. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)* (2015).

[24] Gastwirth, J. L. The estimation of the lorenz curve and gini index. *The Review of Economics and Statistics* (1972), 306–316.

[25] Gordon, M., Kim, D., Perkins, J., Gilham, L., Nguyen, N., and Rinard, M. Information-Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2015).

[26] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: Scalable and Accurate Zero-Day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2012).

[27] Hou, T.-W., Chen, H.-Y., and Tsai, M.-H. Three control flow obfuscation methods for java software. *IEE Proceedings-Software 153*, 2 (2006), 80.

[28] Keogh, E., and Mueen, A. *Curse of Dimensionality*. Springer US, Boston, MA, 2010, pp. 257–258.

[29] Kohavi, R., et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (1995).

[30] Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., van der Veen, V., and Platzer, C. ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (2014).

[31] Liu, J., Wickerson, J., and Constantinides, G. A. Loop splitting for efficient pipelining in high-level synthesis. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2016).

[32] Lockheimer, H. Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, February 2012.

[33] Louppe, G., Wehenkel, L., Sutera, A., and Geurts, P. Understanding variable importances in forests of randomized trees. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)* (2013).

[34] Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., and Stringhini, G. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).

[35] Moser, A., Kruegel, C., and Kirda, E. Limits of static analysis for malware detection. In *Proceedings of the ACM Computer Security Applications Conference (ACSAC)* (2007).

[36] Pomilia, M. A study on obfuscation techniques for android malware, 2016.

[37] Qi, Y. *Random Forest for Bioinformatics*. Springer US, Boston, MA, 2012, pp. 307–323.

[38] Rastogi, V., Chen, Y., and Jiang, X. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the ASIA ACM Symposium on Computer and Communications Security (ASIACCS)* (2013).

[39] Reina, A., Fattori, A., and Cavallaro, L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April* (2013).

[40] Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the ACM Symposium on Applied Computing (SAC)* (2013).

[41] Tam, K., Khan, S., Fattori, A., and Cavallaro, L. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2015).

[42] Viennot, N., Garcia, E., and Nieh, J. A Measurement Study of Google Play. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2014).

[43] Wei, T., Mao, J., Zou, W., and Chen, Y. A new algorithm for identifying loops in decompilation. In *Proceedings of the International Conference on Static Analysis (SAS)* (2007).

[44] Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., and Enck, W. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the International International Conference on Software Engineering (ICSE)* (2015).

[45] Zheng, M., Lee, P. P., and Lui, J. C. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2013).

[46] Zheng, M., Sun, M., and Lui, J. C. Droidanalytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Proceedings of the International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (2013).

[47] Zhou, Y., and Jiang, X. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2012).

[48] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2012).