
# Improving Accuracy of Android Malware Detection with Lightweight Contextual Awareness

Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji,
Simon Pak Ho Chung, and Wenke Lee
Georgia Institute of Technology

## ABSTRACT

In Android malware detection, recent work has shown that using contextual information of sensitive API invocation in the modeling of applications is able to improve the classification accuracy. However, the improvement brought by this context-awareness varies depending on *how* this information is used in the modeling. In this paper, we perform a comprehensive study on the effectiveness of using the contextual information in prior state-of-the-art detection systems. We find that this information has been "over-used" such that a large amount of non-essential metadata built into the models weakens the generalizability and longevity of the model, thus finally affects the detection accuracy. On the other hand, we find that the *entrypoint* of API invocation has the strongest impact on the classification correctness, which can further improve the accuracy if being properly captured. Based on this finding, we design and implement a *lightweight*, circumstance-aware detection system, named "PIKADROID" that only uses the API invocation and its entrypoint in the modeling. For extracting the meaningful entrypoints, PIKADROID applies a set of static analysis techniques to extract and sanitize the reachable entrypoints of a sensitive API, then constructs a frequency model for classification decision. In the evaluation, we show that this slim model significantly improves the detection accuracy on a data set of 23,631 applications by achieving an f-score of 97.41%, while maintaining a false positive rating of 0.96%.

## CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**; *Mobile platform security*;

## KEYWORDS

Malware detection, Android Security

**ACM Reference Format:**
Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji, Simon Pak Ho Chung, and Wenke Lee . 2018. Improving Accuracy of Android Malware Detection with Lightweight Contextual Awareness. In *Proceedings of 2018 Annual Computer Security Applications Conference, San Juan, PR, USA, December 3–7, 2018 (ACSAC'18),* 12 pages.
https://doi.org/10.1145/3274694.3274744


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

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274744

## 1 INTRODUCTION

The Android operating system has become the most popular mobile OS with two billion monthly active devices [24]. Unfortunately, the popularity of the Android OS and its feature-rich environment have made it a popular target for malicious developers. Sixteen million new malware samples were identified in the final quarter of 2017 alone [33]. In spite of the counter-measurements by the Anti-Virus industry as well as Google, Android malware is still widely active and making multi-million dollar profits. For example, LokiBot, a banking trojan, has created over two million dollars in revenue by attacking the customers of 121 international and domestic banks [33].

Due to the significance and popularity of the Android OS, there has been a large body of research for Android malware detection [2, 3, 5, 7, 10, 11, 15, 16, 21, 31, 52, 53, 55]. Several prior approaches have tried to provide efficient and scalable malware detection by analyzing the permissions used by an application [15, 22, 55]. Unfortunately, the set of permissions only capture what capabilities an application has but not how they are used so, they cannot be used to accurately identify malware. Moreover, benign applications tend to request more permissions than necessary [14], which causes a high false-positive rate. To address this, several systems have proposed to instead check the usage of critical APIs in the Android framework [2, 5, 21]. Using critical APIs provides a more complete, fine-grained perspective of the application's behavior compared to permissions because APIs cover a more complete set of sensitive information [39]. Also, APIs have one-to-one associations with each sensitive source while each permission usually includes more than one API capabilities. However, API-based systems must distinguish between how sensitive APIs are used in malicious verses benign applications. Therefore, prior approaches prune out many sensitive APIs that are found to be common in benign and malicious apps which affects the accuracy of classification. For instance, DroidAPIMiner[2] does not consider the `TelephonyManager.getDeviceID` API to be sensitive because it is used by benign and malicious apps with similar frequency. Including APIs like this results in higher false negatives rates. Instead, we find the *lightweight context* in which a behavior is executed helps reveal the true intension. For example, when a user presses the `cancel` button on a dialog, malicious applications are 11 times more likely to access a device's ID via the `TelephonyManager.getDeviceID` API than benign ones. Leveraging such contextual information makes the classification of APIs more accurate, which has the chance to improve the malware detection accuracy.

Several prior works, such as AppContext [52] and Droidsift [53] propose to embed the contextual factors of the API invocation in the modeling and classification. For example, AppContext ties a

variety of activation events, and behavioral information to the sensitive API invocation, while Droidsift constructs a dependency graph for modeling fine-grained contextual information. However, the modeling of many contextual factors becomes challenging because it requires intensive labeling effort and the detection accuracy is degraded when too much non-essential contextual information is included [41]. Additionally, it has been shown that context-based systems, like AppContext, suffer from overfitting because they are too tailored towards the training samples [51]. This issues makes these models perform poorly on new samples of malware, especially from different time periods. To address this, leveraging an abstraction-based approach to generalize program behavior has been proposed. Prior work has shown this can reduce overfitting and increase the lifetime of the trained model [31]. However, we find that one limitation of abstracted-based approaches is that they incur significant context loss which can reduce the performance of the model. Through our experimental evaluation, we find that prior abstraction-based approaches cannot maintain enough context to identify malware that is near the decision boundary.

We rethink the necessity and effectiveness of using contextual information and argue to reduce the amount of contextual factors which are considered and focus only on the most influential factors which we define as *lightweight context*. The goal of lightweight context is to create a more generalizable, contextual representation of the Android app which is less prone to overfitting, as this is a limitation of prior context-based systems [52, 53]. We argue that lightweight context can create this more generalizable representation of the behavior of an Android application which is not tailored to a specific malware family. To validate this argument, we first study the composition of contextual information extracted by existing approaches and evaluate their effectiveness in the modeling and classification of Android applications. The results of our study show that compared to other contextual factors used in prior work [52], the entrypoint provides the most informative features for classification. An entrypoint is defined as a callback that is defined by the application, but called by the framework in response to some event involving the user, the device, or the environment.

We implement a lightweight, context-aware malware detection system for Android apps and evaluate it against previous state-of-the-art detection systems to show the performance improvement using datasets spanning multiple years. Our proposed system, PIKADROID, uses lightweight context to achieve highly-accurate and efficient Android malware detection. PIKADROID develops contextual awareness by answering the question of *how* a sensitive API is invoked in a much *simplified* way. PIKADROID leverages this awareness to infer when sensitive behavior is being used inappropriately which essentially distinguishes legitimate uses of sensitive behavior from malicious intent. Specifically, PIKADROID discovers what Android entrypoint is used to invoke the particular sensitive API, applies a set of reachability and frequency analysis techniques for refining, and then learns how to distinguish under what circumstances malicious applications invoke sensitive behavior. By leveraging lightweight, context-based Android malware detection, PIKADROID is capable of detecting Android malware samples with a f-score of 97.41% and a false positive rate of 0.96%. Next, we compare PIKADROID to prior abstraction-based approaches and find that PIKADROID can outperform prior approaches [31] in classifying malware samples that are 1-5 years older than the training samples.

This paper makes the following major contributions:

- **Case study of context-aware, detection approaches.** We study the effectiveness of existing context-aware, modeling and classification approaches and show these models may have too heavy contextual information which results in lower accuracy and labeling complexity. Additionally, we find the pair of entrypoint and sensitive API is the most informative contextual feature for distinguishing legitimate and illegitimate uses of sensitive behavior.
- **A lightweight, context-aware approach.** We build a lightweight, context-aware system, PIKADROID, that considers the probability of the sensitive behaviors conditioned on the current entrypoint using reachability and frequency analysis.
- **Comprehensive Evaluation.** Our evaluation shows sizable improvements of accuracy over existing approaches. We show that PIKADROID achieves an f-score of 97.41% while maintaining a false positive rating of 0.96% when detecting malware from the same time period. Specifically, we find that compared to prior frequency-based approaches like DroidAPIMiner [2], PIKADROID achieves a 2.2x reduction in false-positives on average. Next, we compare PIKADROID to prior abstraction-based approaches and find that PIKADROID outperforms prior approaches [31] in classifying malware samples that are 1-5 years older than the training samples.

**Paper Organization.** The rest of the paper is organized as follows: §2 discusses the motivation for lightweight context and provides a background on the issues related to context-based systems. §3 and §4 presents the design and implementation of PIKADROID respectively. §5 discusses the dataset and §6 provides the results from our extensive evaluations. Finally §7 discusses the related work and §8 concludes.

## 2 RETHINKING CONTEXTUAL AWARENESS

In this section, we motivate our work by demonstrating existing issues related to context-based, malware detection systems [51–53]. Next, we demonstrate the benefits of *lightweight context* and its ability to generalize an application's behavior.

### 2.1 Less Effective Contextual Information

Some current context-based, malware detection systems embed an app's contextual information into a graph to be used later for classification [53]. One limitation of context, graph-based approaches is they use too much contextual information and create unique features that cause detection to be less effective. Because of this, dependency graphs have been shown to be extremely effective at detecting similar variants of the same malware-family, but they are not good at distinguishing the exact dependencies in a graph that define what truly makes this behavior malicious. This leads to contextual-confusion, which occurs when non-informative dependencies become interwoven with the malware's essential behavior.

*2.1.1 Non-essential Dependencies.* Essential dependencies are the contextual dependencies that provide information related to the malware's core behavior, while residual dependencies are contextual dependencies that are non-essential to the malicious behavior

| Method Type | Activity % (Entrypoints) | Receiver % (Entrypoints) | Intent_params% | Service % (Entrypoints) | Database % | Device % | UI % (Entrypoints) | Network % |
|---|---|---|---|---|---|---|---|---|
| CLASS_LOADING | **47** | **8** | 3 | **13** | 4 | 7 | **12** | 1 |
| REFLECTION_METHOD | **39** | **5** | 10 | **3** | 7 | 9 | **8** | 10 |
| ACCESS_COARSE_LOCATION | **22** | **0** | 20 | **11** | 1 | 15 | **3** | 16 |
| ACCESS_NETWORK_STATE | **27** | **2** | 6 | **15** | 4 | 13 | **11** | 14 |
| ACCESS_WIFI_STATE | **34** | **9** | 9 | **21** | 0 | 7 | **3** | 9 |
| BROADCAST_STICKY | **43** | **2** | 2 | **30** | 0 | 0 | **10** | 0 |
| CAMERA | **68** | **0** | 0 | **0** | 0 | 0 | **20** | 0 |
| CHANGE_WIFI_STATE | **21** | **12** | 9 | **33** | 0 | 1 | **11** | 4 |
| DISABLE_KEYGUARD | **21** | **0** | 0 | **79** | 0 | 0 | **0** | 0 |
| GET_ACCOUNTS | **59** | **1** | 4 | **7** | 0 | 0 | **16** | 0 |
| INTERNET | **43** | **6** | 5 | **12** | 2 | 6 | **13** | 2 |
| READ_PHONE_STATE | **21** | **2** | 4 | **35** | 5 | 16 | **5** | 6 |
| RECORD_AUDIO | **32** | **1** | 0 | **63** | 0 | 0 | **3** | 0 |
| RESTART_PACKAGES | **51** | **5** | 3 | **6** | 1 | 3 | **12** | 0 |
| VIBRATE | **43** | **4** | 6 | **20** | 1 | 2 | **12** | 3 |
| WAKE_LOCK | **44** | **4** | 4 | **13** | 3 | 3 | **16** | 1 |
| WRITE_SETTINGS | **53** | **1** | 8 | **4** | 4 | 1 | **12** | 3 |

**Table 1: A detailed report of the findings in our case study on contextual factors. This table describes the relation between sensitive API categories and contextual factors. Each row represents a API category and each column represents a Context Category, and the value represents feature importance. The feature importance represents the predictive power a feature has in a learning problem (higher scores are better).**

and in many cases represent minor implementation details or generic program functionality.

These non-essential dependencies have significant security consequences when they are used in the feature space. For example, to work around graph-based classification, the malware developer could insert benign routines into their malware to intentionally confuse the classifier. This is effective because the resulting malware would contain more benign behavior than suspicious behavior, bypassing detection. In practice, like normal applications, malware developers also rely heavily on code reuses [10], which causes a classifier's training dataset to contain many samples belonging to the same family. Naturally, this will make the non-informative features reoccur, misleading the classifier into making security-centric decisions based on residual features that aren't necessarily related to the behaviors that the classifier intends to vet. Prior work has shown that the natural redundancy that occurs in malware training sets combined with introducing non-informative features into the feature space leads to downstream classifiers being susceptible to making discriminative decisions based on non-essential features [41]. Instead of including all the dependencies involved in the API invocation, we argue to focus on the lightweight "backbone" of the behavior. Through extensive evaluation, we show this lightweight model achieves better accuracy and is more resilient to the context confusion issue.

*2.1.2 Family-Specific Signatures.* Prior context-based systems add a large amount of context related to the specific app which results in a fine-grain representation of the app's behavior. However, this graph essentially becomes a signature for that particular malware family instead of a more general behavioral profile. Models categorized by family-specific signatures in fact weaken the main goal of these systems which is to rely on generic features to detect unknown variants or zero-day attacks. AppContext [52] is a state-of-the-art Android malware detection system that distinguishes illegitimate and legitimate uses of sensitive behavior by tying context-factors, such as activation events, and behavior information, to security-sensitive behaviors. AppContext attaches these contextual factors to a security-centric event to identify the intent behind the behavior. This approach allows AppContext to provide a fine-grained view of the combination of contextual factors that influenced the execution of a security-sensitive behavior. Unfortunately, since so many possible combinations of contextual factors are possible, AppContext requires intensive, manual labeling of sensitive behaviors before it can be trained. Additionally, even though the authors are domain experts, in several cases sensitive behaviors were mislabeled.

One prior approach which has been used to avoid the intensive labeling process is frequency analysis [2, 53]. For example, DroidAPIMiner [2] leverages frequency analysis to determine which sensitive APIs are more common in malicious applications. However, in order for frequency analysis to be effective, the features needs to be generalizable. This is required because the discriminating factor of frequency analysis depends on features frequently reoccurring in the same class. Unfortunately, since AppContext uses so many contextual factors in its decision making process, applying frequency analysis in an straight-forward manner would be difficult. This is because as the set of contextual dependencies grows larger, the likelihood of it being unique increases. So, although we agree that context is necessary to make informative decisions about an application's intent, it is also necessary to distinguish better information from non-informative factors. This approach allows context-based systems to identify the key contextual factors that make up the backbone of the malware's behavior while also avoiding non-informative dependencies in order to maintain the generality of the feature space.

## 2.2 Case Study: Identifying Informative Context Factors

In this case study, we collect 54,969 labeled sensitive behaviors to study the dependency relationships between Android sensitive APIs and the corresponding contextual factors. We extract the contextual factors for each behavior with a state-of-the-art, context-based system, AppContext [52]. AppContext uses static program analysis to capture the events and conditions that each security-centric API is dependent on. By extracting this context, AppContext determines under what events and conditions this security-sensitive event occurred. Next, AppContext uses these contextual factors as features

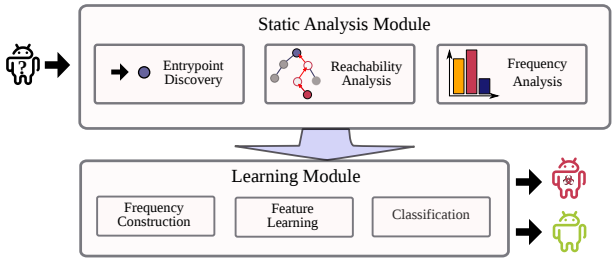


**Figure 1: Visualization of the four stages of analysis of PIKADROID. To classifying a unknown app. (a) PIKADROID each app first goes through the reachability module, which identitfies the sensitive APIs invokable from each entry point. (b) Next, the reachability results are passed into the resolution module, where Class Hierarchy Analysis (CHA) is applied to identify what classes were extended. (c) Feature Construction (d) PIKADROID determines if the application is malicious or benign.**

to classify whether the sensitive API is used inappropriately or not. The intuition here is that the events and conditions under which a security-sensitive method is invoked differ in benign and malicious applications.

In this study, we address the question: *what are the informative, contextual factors that highly distinguish between legitimate and illegitimate uses of sensitive behavior?* To answer this question, first sensitive behaviors are categorized into several sets. We leverage the Android permission model to do the categorization. For example, `HttpClient.execute` belongs to the `INTERNET` category. Unguarded APIs, such as `ClassLoader.loadClass`, are then placed in categories based on their functionality. For example, the `CLASS_LOADER` category contains methods related to dynamic code loading. AppContext also creates contextual categories such as location, system, and network. Contextual categories for entrypoints were created by grouping similar entrypoints into four categories `Activity`, `Service`, `Broadcast Receivers`, `UI`.

After completing the categorization process, we rank each contextual category based on how influential they are on determining if the behavior is benign or malicious using Ridge Regression [23]. The results are shown in Table 1 (the higher the percent, the more important the feature). The remaining categories are discussed in §6.2.4. The rank and importance of each context category implies which contextual factors provide informative contextual information and what factors are potentially non-informative. In almost all cases, the `ENTRYPOINT` categories were ranked the highest. Following the entrypoint category is the `NETWORK` category. Therefore we conclude that the entrypoint is the strongest factor, which indicates its potential in being the most discriminative and generalizable power in classification. This finding motivates us to use it in our proposed lightweight contextual model.

## 2.3 Calling for Lightweight Context Dependencies

Identifying under what circumstances a behavior occurred helps differentiate malicious and benign applications because the additional context helps distinguish between malicious and benign uses of sensitive behavior. We develop lightweight, circumstance awareness by identifying what entrypoint was used to execute a sensitive API. The choice to use the entrypoint is based on the stark differences we find in what entrypoints malicious and benign applications use to invoke sensitive behavior. For example, in Table 2 we provide the likelihood of a malicious application executing a HTTP client from different entrypoints. Despite this API being used by more benign apps, malicious applications are 6.89 times more likely to invoke it from the `Service.onCreate` entrypoint. Based on this observation, we build a model that includes the entrypoints of the sensitive API invocation, improving the accuracy of malware detection.

| Entrypoint | Risk Score |
|---|---|
| `Unconditioned on entrypoint (API only)` | 0.90 |
| **`Activity.onDestroy`** | **15.15** |
| **`Activity.onRestart`** | **14.46** |
| **`DialogInterface.onClick`** | **3.39** |
| `Preference.onPreferenceChange` | 1.77 |
| **`Service.onCreate`** | **6.89** |
| **`Service.onStart`** | **6.20** |
| `webkit.onDownloadStart` | 1.03 |

**Table 2: The ratio of of how frequently malicious applications executed a HTTP Client from different entrypoints in malicious versus benign apps as well as not conditioned on any entrypoint**

# 3 PIKADROID

## 3.1 Overview

We propose a system that uses *lightweight*, circumstantial awareness for Android malware detection. We depict the architecture of PIKADROID in Figure 1. First, the static module extracts sensitive behaviors and the lightweight context in which they occur. The output of this step is the set of all (*entrypoint*, *target API*) pairs extracted for each app in a training set. After the static analysis is complete, PIKADROID uses the extracted pairs to learn under what circumstances the invocation of a sensitive API should be considered malicious. Similar to prior approaches [2], we assume pairs that are found more in malicious applications are naturally more malicious. This allows PIKADROID to identify when behaviors that are commonly used by both benign and malicious applications are being used inappropriately. After the frequency analysis is completed, a risk score, $r_{e,t}$, is assigned to each pair (*entrypoint*,*target API*). The risk scores are then used to create a feature vector for each app. Finally, PIKADROID uses the apps in the training set to train PIKADROID's classifier. We test several machine learning classifiers for comparison, and find that a RandomForest Classifier [29] performs the best (refer to §6.5 for detailed performance evaluation). After completion of the training phase, PIKADROID's model can be used to accurately classify benign and malicious apps.

## 3.2 Static Analysis Module

PIKADROID's static module applies reachability analysis starting from each entrypoint used by an app to a set of configurable, targeted, sensitive APIs methods. The goal of the static analysis module is to identify what sensitive behaviors are used by an application, and under what lightweight context this behavior is invoked. To identify what set of behaviors in the Android framework should be considered sensitive, we leverage SUSI [39] to identify a comprehensive lists of sensitive source and sinks and Pscout [6] to identify permission-guarded APIs. Additionally, the Android framework leverages over 20,000 different callbacks that could potentially be

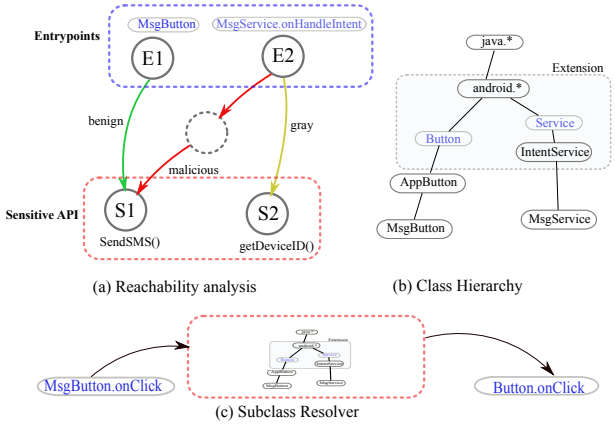


**Figure 2: Detailed illustration of PIKADROID's static-analysis module. The static analysis phase applies (a) reachability analysis from each entrypoint in the application to identify what sensitive APIs are invokable from each entrypoint. Next, the (b) class hierarchy is used by the (c) subclass resolver to generalize the feature space and remove application syntax.**

```
1  public class MsgService extends IntentService {
2
3    /* The entrypoint invoked when an intent is received. */
4    protected void onHandleIntent(Intent intent) {
5      ...
6      if (!checkID(getApplicationContext()) {
7        collectPayment();
8      }
9      ...
10   }
11
12    /* Sends premium text message. */
13    public void collectPayment() {
14     ...
15     SmsManager.getDefault().sendTextMessage(...);
16   }
17
18   /* Check if this is a real device. */
19   public static boolean checkID(Context context) {
20     ...
21     deviceId = telephonyManager.getDeviceId();
22     return deviceId.equals("000000000000000");
23   }
24 }
```

**Listing 1: A sms trojan checking the device ID and only sending a premium text message if it is a real device.**

used as entrypoints [9]. PIKADROID refines this list into a set of sensitive entrypoints including component lifecycle methods, system level callbacks, UI callbacks, and security-sensitive callbacks. Our static analysis module uses these entrypoint and API lists to produce a set of all pairs (`entrypoint`, `targetMethod`) found in the application, which will be passed to the learning module. The following sections detail each step in the static analysis pipeline.

**Preprocessing.** PIKADROID first pre-processes the apps for the static analysis. App's `apk` files are disassembled and unpackaged using `apktool` [47]. The `dalvik` classes are retargeted to Java bytecode using `Dare` [35], and the bytecode is transformed into Wala's internal Intermediate Representation [19] for further analysis.

Next, PIKADROID builds a callgraph for each app being analyzed. In order to generate a precise callgraph, a starting point for the application must be defined. Android apps are event driven and therefore do not have a generic `main()` method which is present in standard Java programs. Instead, apps have components which begin executing in response to events from the device and the user. These components are implemented by extending a pre-defined component class provided by the Android framework. Each component is then statically registered in `AndroidManifest.xml` so the device knows what component lifecycle method to call in response to certain events (PIKADROID identifies dynamically-registered broadcast receivers). To ensure that the callgraph generation algorithm finds all possible paths through the app, we create a dummy main method which contains all possible lifecycle transitions for each component registered in the app's manifest. This dummy method creation follows an approach similar to FlowDroid [20]. Next, to model inter-component communication (ICC), we leverage an approach similar to Epicc [36] to append ICC edges onto the callgraph.

**Entrypoint Discovery.** On top of the callgraph, we use an entrypoint discovery method similar to prior work [20, 48, 53] to discover all the entrypoints in the callgraph which will be used in defining lightweight context. After the entrypoints have been identified, the next step is to use reachability analysis from each entrypoint to identify what sensitive behavior can be invoked. For each entrypoint, PIKADROID does reachability analysis to identify any callsite of targetable methods. If a callsite is reachable from a particular entrypoint, a pair, (`entrypoint`, `targetMethod`), is generated for later analysis. At this stage, the entrypoint signatures are defined in application specific classes which extend the base component classes in the Android SDK. For example, in Listing 1, the `MsgService` class has extended the `IntentService` class and defined the `onHandleIntent` callback. The entrypoint `MsgService.onHandleIntent` belongs to a subclass that has extended a class in the Android framework, Additionally, there could be multiple levels of inheritance, as is the case for `MsgButton`, whose parent class is `AppButton`. If a application-specific entrypoint signature (`MsgService`) was used to create features for each app, the resulting feature space would consist of sparse feature vectors that included application syntax. Including application syntax into the feature space makes the model more susceptible to overfitting and evasion. Overfitting occurs because the entrypoint signature most likely only belongs to a specific malware family. Additionally, the model becomes more susceptible to evasion because entrypoint signatures are developer defined, and the signature name could be obfuscated and randomized using tools such as ProGuard [28] resulting in features unique to a single app.

**Subclass Resolver.** To remove application syntax, we develop a subclass resolver module that identifies the signature of the base class in the Android framework which the application specific class was inherited from. Figure 2(c) shows PIKADROID's subclass resolver, which takes in a reference to the overridden or implemented entrypoint signature and identifies the signature of the original base class in the Android framework that was overridden. The resolution procedure is based on the namespaces provided by Java's classloaders. Specifically, PIKADROID utilizes three classloaders: 1) the *application* classloader that loads the app's code 2) the *primordial* classloader, which is responsible for loading the Java library code (`java.*`), and 3) the *extension* classloader, which is responsible for loading system-wide, platform-specific extension classes

(`Android.*`.) The classloaders form a class hierarchy tree whose leaves correspond to app specific classes and whose nodes towards the root are from the Android library. For example, Figure 2(b), shows a snippet of the class hierarchy tree for `MsgButton`. When the subclass resolver needs to identify which base class is overridden, it performs a backwards traversal of the class hierarchy, from the node belonging to the method signature in the app. The traversal stops when it reaches a class that is provided by the extension classloader. For example, if the input into the subclass resolver was the signature, `MsgService.onHandleIntent` the output would be `IntentService.onHandleIntent`. The final output of PIKADROID's static analysis module is the set of resolved pairs of (`entrypoint`, `targetMethod`) pairs where the `entrypoint's` class belongs to the Android framework These generalized features are then passed to PIKADROID's learning module.

### 3.3 Learning Module

The intuition behind our features is shown in the following simple example. In our dataset, benign and malicious applications used the sensitive method `HttpClient.execute` at similar rates. However, in Table 2, we provide the likelihood of a malicious application invoking the method from different entrypoints. Despite this method being used with similar frequency by both benign and malicious applications, malicious applications were 6.89 times more likely to invoke it from the `Service.onStart` entrypoint and 15.15 times more likely to invoke it from `Activity.onRestart` compared to benign applications. We use this intuition to inform our feature extraction method.

To learn the different lightweight contexts associated with each entrypoint, PIKADROID uses a training set of apps, $T$. For each app $a \in T$, PIKADROID first performs static analysis to extract the set of (`entrypoint`, `targetMethod`) tuples as detailed in the prior subsection. Then, the system partitions $T$ into two sets, $M$ and $B$, where $M \subset T$ is the set of malicious applications and $B \subset T$ is the set of benign applications. Next, we assign a risk score, $r_{e,s}$, to each pair that captures the relative maliciousness of the API, $s$ being invoked from $e$. The risk score is calculated as the ratio of how often the pair $(e, s)$ was found in malicious apps to benign apps. Equation 1 shows the equation PIKADROID uses to calculate this risk value for each pair, $(e, s)$.

$$r_{e,s} = \frac{|\{a|a \in B \wedge (e,s) \in a\}| \times |M|}{|\{a|a \in M \wedge (e,s) \in a\}| \times |B|} \quad (1)$$

The final phase of PIKADROID's analysis is feature construction and model training and testing. The feature vector for each $a \in T$ is $v_a = \langle r_{e,s} : (e,s) \in a \rangle$, the union of risk values for each $(e,s) \in a$. The ground truths are also created in this step and are equal to $0 \; \forall \; a \in B$ and $1 \; \forall \; a \in M$. The results of these tests are presented and discussed §6.4

## 4 IMPLEMENTATION

We implemented a prototype of our system, PIKADROID. The static module was implemented in 8 thousand lines of Java code, based on IBM's WALA Static Analysis Libraries [19]. We leveraged WALA to generate the callgraph, reachability analysis, and entrypoint analysis. The learning module was implemented using 3 thousand lines of Python code, and was based on Scikit-learn [38], a machine-learning Python library.

| Dataset | Time Period | Malicious | Benign | Grayware | Total |
|---|---|---|---|---|---|
| 2010-2012 | 2010 - 2012 | 3,970 | 3,788 | 1,524 | 9,282 |
| 2013-2015 | 2013 - 2015 | 2,158 | 3,596 | 1,325 | 7,079 |
| 2016-2018 | 2016 - 2018 | 2,270 | 5,000 | — | 7,270 |
| **Total** | **2010 - 2018** | **8,398** | **12,384** | **2,849** | **23,631** |

**Table 3: Summary of the datasets used for evaluations.**

## 5 DATASET

The dataset used to extensively evaluate PIKADROID contained malicious apps crawled from over 16 official and third-party Android app marketplaces collected using Androzoo [4]. We crawled the benign apps from Google Play. The dataset contained over 23,631 applications overall, Our final dataset includes 8398 malicious apps, 12384 benign apps, and 2849 grayware apps. Additionally, this dataset spanned a time-period of eight years. Using such a wide range of diverse sources is necessary to conduct a robust evaluation due to the app sampling problem in Android [32]. Next, we leveraged Virus Total [45] for label verification and to identify grayware. To be able to assess how well PIKADROID handles concept drift [25], we manually partitioned the dataset into three different time periods, 2010-2012, 2013-2015, 2016-2018. We used the date when the app was signed to assign an app to each set. A detailed description of each partition is shown in Table 3.

## 6 EVALUATION

Our evaluation addresses the following research questions:

- How effective is PIKADROID in improving the accuracy of Android malware detection?
- How does the lightweight circumstantial awareness model perform compared with existing state-of-the-art systems? Particularly, how does PIKADROID handle "concept drift", a key challenge in Android malware detection?
- How robust is PIKADROID against obfuscation?
- What is the runtime performance of PIKADROID?

We perform a comprehensive evaluation covering a variety of cases in verifying the classification of Android malware [25, 26]. First, we focus on the effectiveness of PIKADROID in terms of accuracy, precision, recall, f1-score, and false-positive rating. Second, we perform side-by-side comparisons in terms of detection accuracy between PIKADROID and three prior state-of-the-art detection systems: AppContext [52], MaMaDroid [31] and DroidAPIMiner [2]. In particular, we compare how these systems handle "concept drift", a key challenge that can significantly affect detection effectiveness. Third, to test the counter-obfuscation ability of PIKADROID, we use a state-of-the-art obfuscation tool, AVPass [27] to generate obfuscated variants of the malicious samples and test PIKADROID's accuracy when classifying these apps. Fourth, we measure the performance with different datasets spanning multiple years. To evaluate PIKADROID, we use the datasets discussed in §5 and 10-fold cross validation for all tests.

| Dataset | | Accuracy (%) | | Precision (%) | | F1 Score (%) | | Recall (%) | | FP Rating | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Samples** | **Time Period** | **Pika** | MaMa | **Pika** | MaMa | **Pika** | MaMa | **Pika** | MaMa | **Pika** | MaMa |
| Malware vs. Benign | 2010-12 | **97.61** | 94.66 | **98.29** | 97.13 | **97.65** | 94.64 | **97.02** | 92.28 | **1.76** | 2.87 |
| | 2013-15 | **98.43** | 97.55 | **99.24** | 97.55 | **97.89** | 96.70 | **96.57** | 95.87 | **0.44** | 1.45 |
| | 2016-18 | **97.56** | 96.53 | 96.90 | 97.08 | **96.07** | 94.27 | **95.24** | 91.61 | 1.38 | 1.25 |
| | 2010-18 | **98.00** | 95.67 | **98.52** | 95.81 | **97.41** | 94.58 | **96.33** | 93.38 | **0.96** | 2.77 |
| Grayware vs. Benign | 2010-12 | **98.15** | 95.68 | **97.98** | 97.85 | **96.74** | 92.02 | **95.53** | 86.84 | 0.79 | 0.77 |
| | 2013-15 | **96.91** | 92.63 | **97.80** | 91.09 | **94.04** | 85.45 | **90.57** | 80.47 | **0.75** | 2.90 |
| | 2010-15 | **96.83** | 93.22 | **96.92** | 94.86 | **94.15** | 86.78 | **91.54** | 79.97 | **1.12** | 1.67 |
| (Mal-&Grayware) vs. Benign | 2010-12 | 96.99 | 97.55 | **97.87** | 97.55 | **97.44** | 96.70 | **97.03** | 95.87 | 3.06 | 1.45 |
| | 2013-15 | **96.54** | 93.57 | **97.59** | 94.60 | **96.44** | 93.39 | **95.32** | 92.21 | **2.28** | 5.10 |
| | 2010-18 | **97.79** | 92.63 | **98.27** | 93.90 | **97.47** | 92.12 | **96.70** | 90.40 | **1.34** | 5.33 |

**Table 4: Comparison of PIKADROID and MaMaDroid [31] in accuracy, f-score, precision, recall, and false positive rating when being used at the same time period. The training and testing data are from the same time periods. The cases where PIKADROID outperforms MaMaDroid are in bold.**

## 6.1 Effectiveness

Our first goal is to evaluate how well PIKADROID can correctly classify malware during the time periods when it was most prevalent. Our evaluation includes four datasets. Three datasets contain benign and malware samples from the same time-periods: 2010-2012, 2013-2015, 2016-2018. The first three datasets verify that lightweight circumstantial awareness can accurately distinguish between samples that were popular in relatively the same time periods. The final dataset, 2010-2018, contains malware that spans over eight years. This dataset is used to verify that PIKADROID is capable of learning a diverse set of malware samples. This is necessary because the Android OS is heavily fragmented [46], and malware developers may still use old malware samples to attack outdated devices.

The accuracy, precision, f-score, recall, and false positive rating for each experiment are summarized in Table 4. We see that for each dataset, PIKADROID is able to maintain a low false positive rate of under 1.76% for all cases. For malware in the 2013-2015 dataset, PIKADROID is able to maintain a false-positive rate of 0.44%. Maintaining a low false positive rate (FPR) is very important for Android malware detection systems because high FPRs result in more benign samples being misclassified as malware samples. For each misclassified benign sample, a intensive manual process is required to verify the legitimacy of the app. We see that PIKADROID is able to detect 97.02% of the samples in the 2010-2012 dataset. We also evaluated PIKADROID on the larger dataset which contained samples from all three time periods. We found that PIKADROID was able to detect malware with an accuracy of 98.00% when trained with malicious samples from different time periods.

## 6.2 Comparison with Prior Work

*6.2.1 With AppContext.* For modeling, AppContext [52] creates a contextual model of an app based on a variety of factors: environmental-, activation-, and constraint-dependencies. We use Ridge Regression [23] to rank the contextual factors by importance. Performance wise, we find that the use of multiple types of contextual information as AppContext does brings two challenges. First, the capturing of these contextual behaviors is unlikely to scale to a real-world environment. For example, we originally intended to classify all 23,631 applications in our dataset using AppContext. However, it imposes a size cap of 5 MB on the analyzed applications, with larger apps requiring significant manual efforts to extract the contextual features. In addition, we find that in many cases the analysis takes over 4 hours and consumes over 256 GB of physical memory. In comparison, PIKADROID's static analysis is lightweight, and on average it finishes the analysis in one minute per app (refer to §6.5 for detailed performance evaluation). Unfortunately the lack of scalability related to AppContext's method prevented a side-by-side comparison. To provide a comparison of PIKADROID and AppContext an extension of the case study in §2.2 is provided. In the case-study provided in §2.2, the entrypoints, `activity`, `broadcastReceiver`, `service`, and `UI`, were among the highest ranked categories. The second highest ranked category is behaviors related to `NETWORK` dependencies. We summarize the comparison results in Table 1. Therefore, based on this comparison, the model of PIKADROID using entrypoints provides a lightweight and scalable approach to malware detection by avoiding non-informative contextual factors, which limits AppContext's scalability.

*6.2.2 With MaMaDroid.* Another current, state-of-the-art Android Malware detection system is MamaDroid [31]. This system focuses on abstracting sequences of method and API calls found in an app to build a Markov chain model which represents the probabilities of transitioning between method/API calls. In an effort to minimize features unique to a malware family or single app, MamaDroid abstracts each method/APIs signature to its containing package name and creates the transition model from the resulting set. For example, `android.view.KeyEvent` would become `android.view`. This abstraction process is used to be resilient to the frequent changes in the Android Platform and to prevent feature explosion. For comparison, we use the open-sourced part of MaMaDroid [1] for its abstraction and modeling, then implemented the classification portion, which is not provided in its source code using a RandomForest Classifier [29], as described in the original paper. We evaluate both systems on each of our partitioned datasets as well as the combined dataset covering eight years. A detailed comparison of PIKADROID and MaMaDroid in terms of accuracy, precision, f-score, recall, and false positive rating is given in Table 4. When comparing the results of the classification of malware and benign samples in the same time-period, we verify that both PIKADROID and MaMaDroid are effective at detecting Android malware. Overall, PIKADROID outperforms MaMaDroid in every experiment in terms of f-scores. Even though, MaMaDroid maintained lower false positive rate in 3/10 experiments, PIKADROID maintained a slightly better detection rate in these tests. These tests show that context-based, Android malware detection systems can achieve the same high performance as abstraction-based

| Dataset | FP Ratio | FN Ratio |
|---|---|---|
| 2010-2012 | 1.45x | 1.38x |
| 2013-2015 | **3.02x** | 1.24x |
| 2016-2018 | 2.035x | 1.02x |
| 2010-2018 | **2.36x** | 1.29x |

**Table 5: The ratio of false positive and false-negatives of APIMiner to PIKADROID. Column one shows the ratio of false-positives of APIMiner to PIKADROID. Column two shows the ratio of false-negatives of APIMiner to PIKADROID.**

methods, and in some cases achieve higher detection rates while still maintaining low false positive rates. We believe this shows that both abstraction-based and lightweight circumstantial awareness approaches for malware detection can accurately detect malware when the testing and training data are from the same time periods.

The starkest difference in these comparison tests is the classification of grayware. Grayware creates new challenges for classification because its behavior blurs the line between what is malicious and what is benign. For the 2010-2015 dataset, PIKADROID detects 91.54% of grayware, but MaMaDroid only detects 79.97% of the samples. We find that the discrepancy is due to MaMaDroid abstracting method signatures to package names. This difference comes from grayware's reliance on inappropriate UI-driven event-handlers to trigger advertising displays. For example, grayware samples would register `android.view.KeyEvent` callbacks that display ads and access sensitive device information. Since MaMaDroid relies on only using the package name (`android.view`) instead of the full API signature, it loses the necessary context, since it cannot see what callback is used.

*6.2.3 With DroidAPIMiner.* To evaluate how meaningful the contextual information obtained from conditioning the target API call on its corresponding entrypoint is for malware detection, we compare PIKADROID to prior works by implementing a frequency-based approach like DroidAPIMiner [2], which we call APIMiner for the remainder of the paper. Unlike PIKADROID, these systems do not take into consideration the circumstances in which the sensitive API are invoked. To evaluate how much improvement is obtained from using our contextual awareness, we evaluated APIMiner on each dataset discussed in §5. We provide a detailed report of the false positive ratio and false negative ratio of APIMiner to PIKADROID in Table 5. We see that by not leveraging the entrypoint, it can lead to 3.02x times more false-positives.

In Table 5, we show the reduction of false-positives and false-negatives PIKADROID receives when using the pairs of (`entrypoint`, `targetMethod`) compared to using only the sensitive APIs alone. For the 2010-2018 time period, PIKADROID reduces the false positives by 2.36x and the false negatives by 1.29x. This suggests that PIKADROID's lightweight circumstance-aware approach leads to a significant reduction in false positives and false negatives.

*6.2.4 Reducing False-positives.* We provide an additional case study that shows how lightweight circumstantial awareness allows PIKADROID to detect behaviors that have malicious intent that is undetectable when the APIs are considered alone. When PIKADROID is configured to only use sensitive APIs, the malicious sample with the md5sum "e2caa60b08ea474f0812fabac0985a19", from the DroidKungfu [54] malware family is not detected as malware. In Table 6, we show a subset of (`entrypoint`, `targetMethod`) pairs that are present in this sample. The column `Risk Score (T)` shows the risk-score of the sensitive targeted API, $T$, when the targeted API is considered without being conditioned on the entrypoint. The risk score is a comparison of the number of benign samples that use this particular API versus the number of malicious samples which use this targeted API. For example, the sensitive API `DataOutputStream.writeBytes` had a risk-score of 0.256. This means it was only found in 0.256 of the malicious apps. Prior approaches like DroidAPIMiner would find this behavior uncommon in malicious apps, and the sensitive API would be considered noise and be filtered from the feature space [2]. Unfortunately, this approach leads to significant context loss which decreases detection accuracy.

| Entrypoint (E) | Targeted API (T) | Risk Score (E,T) | Risk Score (T) |
|---|---|---|---|
| Service.onStart | FileWriter.write | 3.06 | 1.20 |
| Service.onStart | DataOutputStream. writeBytes | **18.71** | 0.256 |
| BroadcastReceiver. onReceive | Cursor.getString | 2.59 | 1.46 |
| Service.onCreate | TelephonyManager. getDeviceID | **11.05** | 0.401 |

**Table 6: Comparison of risk scores using (`entrypoint`, `targetMethod`), or (E,T) of PIKADROID and target-API-based system APIMiner for the sample "e2caa60b08ea474f0812fabac0985a19", which belongs to the DroidKungfu [54] malware family. By having an outstanding risk score using (E,T), PIKADROID detects this sample while APIMiner does not.**

| Time Period | Obfuscations | TPR% |
|---|---|---|
| 2010-2012 | string encryption, package obfuscation, class obfuscation, identifier mangling, resource injection, permission injection, bytecode transformations, method obfuscation | 92.5 % |

**Table 7: The obfuscations applied to bypass PIKADROID using AV-Pass [27]. The first column represents the time period when the malicious apps are found; the second column lists the obfuscations, and the third column provides the True Positive Rating (TPR%).**

PIKADROID leverages lightweight circumstantial awareness to create a risk score based on the sensitive API and a lightweight representation of the contextual factors involved in the invocation of this security sensitive method. The column `Risk Score (E, T)` shows the risk-score of the sensitive, targeted API, $T$, when it was invoked using the entrypoint $E$. For example, when the targeted API `DataOutputStream.writeBytes` is invoked from the entrypoint `Service.onStart`, it is 18.71x more likely to be from a malicious app.

*6.2.5 Drifting Scenarios.* To test how resilient PIKADROID is to concept drift [25], we created two drifting scenarios and compared the results of PIKADROID, MaMaDroid [31], and APIMiner [2] in each. In the first drifting scenario, each system was trained using the 2013-2015 dataset, and then classified the samples from the 2016-2018 time period. Figure 3(a) shows the results in terms of f-score for PIKADROID, MaMaDroid, and APIMiner. PIKADROID

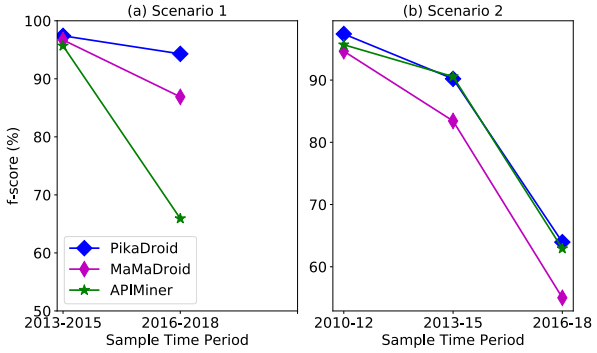


**Figure 3: The results of both drifting scenarios is provided in terms of f score. PIKADROID (blue), MaMaDroid (purple), and APIMiner (green). The left (a) shows scenario 1, which used the 2013-2015 samples as for training PIKADROID. The right (b) shows scenario 2, which used the 2010-2012 samples for training.**

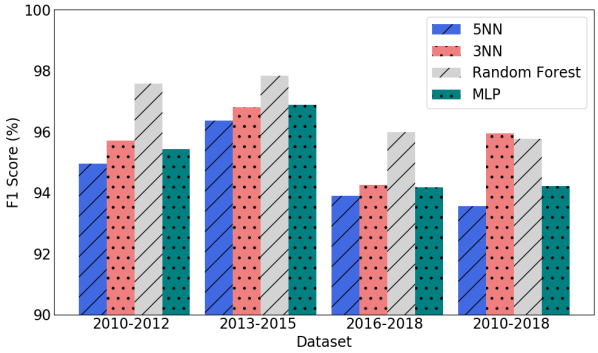


**Figure 4: The f-scores of K-Nearest Neighbor (KNN) ($k = 3, 5$), Random Forest (200 estimators), and a Multi Layer Perceptron with one hidden layer of size 100.**

was able to maintain an f-score of 94.3%. In contrast, MaMaDroid and APIMiner had a f-score 89.89% and 65.92% respectively. We believe this scenario shows that lightweight contextual systems, like PIKADROID, can be extremely effective at generalizing the feature space. This is because PIKADROID aims to learn only the most informative behaviors that form the core of the malware. In scenario two, each system was trained using the 2010-2012 datasets, and then classified samples from 2013-2018. The results are shown in Figure 3(b). Similar to scenario one, PIKADROID and MaMaDroid were able to accurately detect samples 1-5 years older than the training set. However, all three systems performed poorly at detecting apps 6-8 year older. Unlike scenario 1, APIMiner performed suprisingly well, and actually marginally performed better than PIKADROID and MaMaDroid. This leads us to believe that properly evaluating concept-drift is extremely difficult because the evaluation is extremely susceptible to undersampling. This is because evaluating concept-drift requires a diverse dataset over a large time period. Therefore, we believe this evaluation technique may not be the most effective approach at measuring concept drift. However, since it is similar to prior approaches [31], we believe the results contains useful information. However, we believe, a more formal approach, such as Transcend [25] is necessary. Finally, we also believe these two scenarios show that both abstraction-based and contextual-based systems can improve a model's lifetime by creating more generalizable feature spaces. Both MaMaDroid and PIKADROID perform well in both scenarios when the testing samples are 1-5 years older than the training samples.

## 6.3 Robustness

In this section, we discuss experiments used to extensively evaluate PIKADROID's features against state-of-the-art obfuscations tools. We also discuss obfuscations that could potentially hinder PIKADROID's feature set.

**Obfuscated Malware.** In order to evade anti-virus scanners and learning-based systems, malware authors leverage obfuscation techniques to make analysis and reverse engineering more difficult [27, 28, 40, 43]. To identify how robust PIKADROID is to obfuscation, we leverage AVPass [27], an open-source obfuscation platform that is capable of rule and feature inference. This allows malicious developers to tailor combinations of obfuscations techniques to bypass learning-based malware detection systems including PIKADROID. We first train PIKADROID on Android applications in the 2010-2012 dataset. Next, we use AVPass to create unknown obfuscated variants of samples found in the same time period. After the obfuscation process is completed for all apps, we use PIKADROID's trained model to classify them as benign or malicious. We find that PIKADROID accurately detects malicious applications, even when deep-semantic obfuscated apps are used for testing. The results of our experiments are given in Table 7. Eight obfuscations are applied by AVPass, including bytecode transformation, identifier mangling, and class obfuscation. However, despite these obfuscations PIKADROID is able to still detect the unknown variants with a true positive rating 92.5%. We attribute PIKADROID's ability to be robust to its reliance on semantic-based features instead of application syntax like prior approaches [15, 16]

**Limitations.** We see two possible limitations related to the feature space used by PIKADROID. The first limitation is entrypoint obfuscation. Yang et. al. showed possible attacks against context-based systems using phylogenetic analysis [8]. This attack launches an inter-component transplantation attack [50], which transplants the malicious behavior into a different application component. Unfortunately, this attack requires an isolated component (`broadcastReceiver`), which could be easily detected because malicious applications often invoke sensitive APIs from broadcast receivers. PIKADROID is subject to a second limitation such that its contextual factors may be more susceptible to Java reflection and dynamic code loading compared to prior context-based systems [52, 53]. This is because these systems tailor the contextual factors to specific malware families, which allows them to identify inappropriate uses of Java reflection and dynamic code loading. For example, AppContext [52] accurately detects the misuse of Java Reflection and Dynamic Code loading by attaching activation-, control-, and system- dependencies to the invocation of a security sensitive method. Unfortunately, AppContext is only usable on apps that are smaller than 5 MB.

| Dataset | Runtime (s) | Callgraph Nodes | Callgraph Edges |
|---|---|---|---|
| 2010-2012 | 34.4 | 4,728 | 14,627 |
| 2013-2015 | 46.3 | 5,621 | 17,042 |
| 2016-2018 | 55.8 | 5,921 | 18,095 |
| 2010-2018 | 49.4 | 5,584 | 16,989 |

**Table 8: The performance metrics of PIKADROID's static analysis in terms of average runtime, callgraph nodes, and callgraph edges for the 2010-2012, 2013-2015, 2016-2018, and 2010-2018 datasets.**

## 6.4 Classification Models

In order to verify the effectiveness of PIKADROID's lightweight contextual approach independent of the effect from the classification, we conduct a set of experiments using different machine learning classifiers to determine whether the high performance seen in the previous evaluations is a result of a particular type of model or the features themselves. Specifically, we run PIKADROID using K Nearest Neighbors (KNN) with 3 and 5 neighbors as well as a Random Forest (RF) [29] and a Multi-Layered Perceptron (MLP) [42]. For the RF, we have 200 estimator trees and for the MLP we have a `tanh` activation with adam solver and one hidden layer with 100 neurons. Our results, in terms of f-score, are shown in Figure 4. All of the models we test achieve a f-score of 93% or higher on all of the datasets. From these results, we conclude that the high performance of PIKADROID is not simply a result of using a particular classifier, but instead from the lightweight, contextual features extracted from each application.

## 6.5 Performance

We perform the evaluation experiments for PIKADROID on a server with 64 Intel Xeon E7-4820 CPUs running at 2.00GHz and 128 GB of physical memory. For the static analysis module, we configure PIKADROID to build at most 10 callgraphs in parallel. In Table 8, we show the average runtime for PIKADROID's static analysis module. On average, PIKADROID requires less than one minute to analyze each application and 91% of the time analyzing each application is spent on the callgraph building phase. The reachability analysis takes the remaining 9% of the time. We observed that applications from the 2016-2018 time period takes 25% more time to analyze compared to apps from the 2010-2012 period. However, since the runtime is still under one minute, we believe PIKADROID is scalable to analyze large-scale markets. The training of PIKADROID's learning module takes less than ten hours. The majority of the performance is related to parsing the output of the static analysis and can be significantly optimized. Finally, the classification phase is dependent on the model, but we found that all models finished in under 3 hours on our entire, eight-year dataset.

# 7 RELATED WORK

**Android Malware Detection.** There has been many proposed solutions to Android malware analysis that rely on static [2, 3, 5, 7, 10, 15, 16, 21, 22, 31, 52, 52, 53, 55] or dynamic [11, 12, 30] analysis. Drebin [5] uses lightweight static analysis to provide a on-device approach to malware detection. DroidAPIMiner [2] uses frequency analysis to identify critical sensitive APIs that are commonly used by malicious apps. Unfortunately, [2] suffers from feature explosion because it cannot generalize its feature space and achieve its goal of being an on-device detection system. DroidAPIMiner uses frequency analysis to identify critical sensitive behaviors, then uses the set of critical APIs for classification. However, frequency based systems that do not leverage context incur higher false-positive and lower detection rates than PIKADROID. Finally, one promising approach to Android malware detection is MaMaDroid [31]. MaMaDroid leverages sequences of abstracted method calls to create a probabilistic representation of program behavior. This approach creates a more general model that is more robust to unknown variants of malware and new malware families. One limitation of this approach is that it can suffer from context loss, due to the natural loss of information that occurs when abstracting the program semantics into a more coarse-grained representation.

**Context-based Systems.** There has been many systems for identifying contextual dependencies for Android malware using static analysis [17, 18, 34, 37, 51–53]. AppContext [52] extracts contextual factors and activation dependencies to identify malicious behaviors. DroidSift [53], Apposcopy [17], Astroid [18], and Enmobile [51] rely on contextual API dependencies to create complex dependencies that rely on deep-semantics to create robust signatures for malware. Unfortunately, one limitation of all prior approaches that PIKADROID does not have is they cannot generalize their feature space. Otherwise, they would not be able to create the fine-grained signatures they need in order to achieve their main goal of identifying as many possible unknown variants of known malware. Finally, Dark Hazard [37] also leverages static analysis to build context leveraging constraint-relationships. It uses these relationships to identify hidden-sensitive operations, which is orthogonal to PIKADROID's goals. One limitation of PIKADROID is that it cannot identify heavily packed code. In order to addresses this, we could leverage dynamic analysis tools, such as DroidScope [49], DroidUnpack [13], or CopperDroid [44] to dynamically extract contexts. However, we consider this to be out-of-scope of this work.

# 8 CONCLUSION

In this work we present an approach to Android malware detection based on lightweight contextual awareness. We implemented a *lightweight* context-based system, PIKADROID, and completed an extensive evalution, and our evalution shows sizable improvements of accuracy and scalability over existing approaches.

# 9 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful and informative feedback. This research was supported by the ONR under grants N0001409-1-1042, N00014-15-1-2162 and N00014-17-1-2895, and by the DARPA Transparent Computing program under contract DARPA-15-15-TCFP-006. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ONR or DARPA.

# REFERENCES

[1] Mamadroid, Oct. 2017. https://bitbucket.org/gianluca_students/mamadroid_code.
[2] AAFER, Y., DU, W., AND YIN, H. Droidapiminer: Mining api-level features for robust malware detection in android. In *9thInternational Conference on Security and Privacy in Communication Systems (ICST)* (Sydney, Australia, 2013).
[3] ALLEN, J. L. pdroid. *Master's Thesis, University of Tennessee* (2016).
[4] ALLIX, K., BISSYANDÉ, T. F., KLEIN, J., AND LE TRAON, Y. Androzoo: Collecting millions of android apps for the research community. In *Proceedings*

*of the 13th International Conference on Mining Software Repositories (MSR)* (Austin, TX, USA, May 2016).

[5] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., RIECK, K., AND SIEMENS, C. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2014).

[6] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: analyzing the android permission specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)* (Raleigh, NC, USA, Oct. 2012).

[7] AVDIIENKO, V., KUZNETSOV, K., GORLA, A., ZELLER, A., ARZT, S., RASTHOFER, S., AND BODDEN, E. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)* (Florence, Italy, May 2015).

[8] BAXEVANIS, A. D., AND OUELLETTE, B. F. *Bioinformatics: a practical guide to the analysis of genes and proteins*, vol. 43. John Wiley & Sons, 2004.

[9] 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 2015 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2015).

[10] CHEN, K., WANG, P., LEE, Y., WANG, X., ZHANG, N., HUANG, H., ZOU, W., AND LIU, P. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, USA, Aug. 2015).

[11] CHEN, S., XUE, M., TANG, Z., XU, L., AND ZHU, H. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (Xi'an, China, May–June 2016).

[12] DASH, S. K., SUAREZ-TANGIL, G., KHAN, S., TAM, K., AHMADI, M., KINDER, J., AND CAVALLARO, L. Droidscribe: Classifying android malware based on runtime behavior. In *Security and Privacy Workshops (SPW), 2016, (IEEE)* (2016).

[13] 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 2018 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2018).

[14] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[15] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Chicago, IL, USA, Nov. 2009).

[16] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (Chicago, IL, USA, Oct. 2011).

[17] FENG, Y., ANAND, S., DILLIG, I., AND AIKEN, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 19th European Software Engineering Conference (ESEC) / 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (Bergamo, Italy, Aug. 2015).

[18] FENG, Y., BASTANI, O., MARTINS, R., DILLIG, I., AND ANAND, S. Automatically learning android malware signatures from few samples. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2017).

[19] FINK, S., AND DOLBY, J. Wala–the tj watson libraries for analysis, 2012.

[20] FRITZ, C., ARZT, S., RASTHOFER, S., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Highly precise taint analysis for android applications. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Edinburgh, UK, June 2014).

[21] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)* (Hyderabad, India, May–June 2014).

[22] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th ACM International Conference on Mobile Computing Systems (MobiSys)* (Low Wood Bay, UK, 2012).

[23] HOERL, A. E., AND KENNARD, R. W. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics 12*, 1 (1970), 55–67.

[24] I/O, G. Google i/o.

[25] JORDANEY, R., SHARAD, K., DASH, S. K., WANG, Z., PAPINI, D., NOURETDINOV, I., CAVALLARO, L., AND SPA, E. Transcend: detecting concept drift in malware classification models. In *Proceedings of the 26th USENIX Security Symposium (Security)* (Vancouver, BC, Canada, Aug. 2017).

[26] JORDANEY, R., WANG, Z., PAPINI, D., NOURETDINOV, I., SHARAD, K., AND CAVALLARO, L. Poster misleading metrics: On evaluating ml for malware with confidence. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, USA, May 2016).

[27] JUNG, J., JEON, C., WOLOTSKY, M., YUN, I., AND KIM, T. Avpass: Automatically bypassing android malware detection system. In *Black Hat USA Briefings* (Aug. 2016).

[28] LAFORTUNE, E., ET AL. Proguard.

[29] LIAW, A., WIENER, M., ET AL. Classification and regression by randomforest. *R news 2*, 3 (2002), 18–22.

[30] LINDORFER, M., NEUGSCHWANDTNER, M., AND PLATZER, C. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proceedings of the 39th Computer Software and Applications Conference (COMPSAC)* (Taichung, Taiwan, July 2015).

[31] 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. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2016).

[32] MARTIN, W., HARMAN, M., JIA, Y., SARRO, F., AND ZHANG, Y. The app sampling problem for app store mining. In *Proceedings of the 12th International Conference on Mining Software Repositories (MSR)* (Florence, Italy, May 2015).

[33] MCAFEE LABS, L. Mcafee mobile threat report q1, 2018.

[34] NAN, Y., YANG, Z., WANG, X., ZHANG, Y., ZHU, D., AND YANG, M. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2018).

[35] OCTEAU, D., JHA, S., AND MCDANIEL, P. Retargeting android applications to java bytecode. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (Cary, NC, USA, Nov. 2012).

[36] OCTEAU, D., MCDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22th USENIX Security Symposium (Security)* (Washington, DC, USA, Aug. 2013).

[37] PAN, X., WANG, X., DUAN, Y., WANG, X., AND YIN, H. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2017).

[38] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., ET AL. Scikit-learn: Machine learning in python. *Journal of machine learning research 12*, Oct (2011), 2825–2830.

[39] RASTHOFER, S., ARZT, S., AND BODDEN, E. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2014).

[40] RASTOGI, V., CHEN, Y., AND JIANG, X. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (Hangzhou, China, May 2013).

[41] ROY, S., DELOACH, J., LI, Y., HERNDON, N., CARAGEA, D., OU, X., RANGANATH, V. P., LI, H., AND GUEVARA, N. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)* (Los Angeles, California, USA, Dec. 2015).

[42] RUCK, D. W., ROGERS, S. K., KABRISKY, M., OXLEY, M. E., AND SUTER, B. W. The multilayer perceptron as an approximation to a bayes optimal discriminant function. *IEEE Transactions on Neural Networks 1*, 4 (1990), 296–298.

[43] SAIKOA, B. Dexguard.

[44] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)* (Denver, Colorado, Oct. 2015).

[45] TOTAL, V. Virustotal-free online virus, malware and url scanner.

[46] WEI, L., LIU, Y., AND CHEUNG, S.-C. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Singapore, Singapore, Sept. 2016).

[47] WINSNIEWSKI, R. Android–apktool: A tool for reverse engineering android apk files, 2012.

[48] WONG, M. Y., AND LIE, D. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2016).

[49] YAN, L.-K., AND YIN, H. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium (Security)* (Bellevue, WA, USA, Aug.

2012).

[50] YANG, W., KONG, D., XIE, T., AND GUNTER, C. A. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)* (Orlando, FL, USA, Dec. 2017).

[51] YANG, W., PRASAD, M., AND XIE, T. Enmobile: Entity-based characterization and analysis of mobile malware. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, May 2018).

[52] 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 37th International Conference on Software Engineering (ICSE)* (Florence, Italy, May 2015).

[53] ZHANG, M., DUAN, Y., YIN, H., AND ZHAO, Z. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)* (Scottsdale, Arizona, Nov. 2014).

[54] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, USA, May 2012).

[55] 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 19th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, California, USA, Feb. 2012).