Android Malware Detection using Multi-Flows and API Patterns

Feng Shen, Justin Del Vecchio, Aziz Mohaisen, Steven Y. Ko, Lukasz Ziarek
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
{fengshen, jmdv, mohaisen, stevko, lziarek}@buffalo.edu

***Abstract*—This paper proposes a new technique for detecting mobile malware based on information flow analysis. Our approach focuses on the *structure* of information flows we gather in our analysis, and the *patterns* of behavior present in information flows. Our analysis not only gathers *simple* flows that have a single source and a single sink, but also *Multi-Flows* that either start from a single source and flow to multiple sinks, or start from multiple sources and flow to a single sink. This analysis captures more complex behavior that both recent malware and recent benign applications exhibit. We leverage *N-gram analysis* to understand both unique and common behavioral patterns present in Multi-Flows. Our tool leverages N-gram analysis over sequences of API calls that occur along control flow paths in Multi-Flows to precisely analyze Multi-Flows with respect to app behavior. We show the precision of our technique by applying it on 5 different data sets with the total of 6,214 apps—these data sets consist of older generation benign and malicious apps as well as recent benign and malicious apps, showing the effectiveness of our approach across different generations of apps.**

I. Introduction

According to security experts [1], over 37,000,000 malicious applications (apps) have been detected in only a 6-month span in the beginning of 2016. To combat the spread of malicious code, malware detection is crucial. Previous approaches for malware detection have shown that analyzing information flows can be an effective method to detect malicious apps [7, 16, 45]. This is not surprising, as one of the most common characteristics of malicious mobile code is collecting sensitive information from a user's device, such as a device's ID, contact information, SMS messages, location, as well as data from the sensors present on the phone. When a malicious app collects sensitive information, the primary purpose is to exfiltrate it, which unavoidably creates information flows within the app code base.

Many previous systems have leveraged this insight and focused on identifying the existence of *simple* information flows – i.e. considering an information flow as just a (source, sink) pair. A source is typically an API call that reads sensitive data, while a sink is an API call that writes the data read from a source. Using this information, previous approaches determine whether or not an app is malicious based on the presence or absence of certain flows and can achieve 56%-94% true negative rates when applied to known malicious app data sets.

In this paper, we show that there is a need to look beyond simple flows in order to effectively leverage information flows analysis for malware detection. By analyzing recently-collected malware, we show there has been an evolution in malware beyond simply collecting sensitive information and immediately exposing it. Modern malware performs complex computations before, during, and after collecting sensitive information and tends to aggregate data before exposing it. A simple (source, sink) view of information flow does not adequately capture such behavior.

Furthermore, mobile apps themselves have also evolved in their sophistication and in the number of services they provide to the user. For instance, most common apps now leverage a user's location to provide additional features like highlighting points of interest or even other users that might be nearby. Augmented reality apps take a step further, leveraging not only a user's location, but also their camera and phone sensors to provide an immersive user experience. Phone identifiers are now commonly used to uniquely identify users by apps that tailor their behavior to the user's needs. This means that benign apps now use the same information that malicious apps gather. As a direct result, many of the exact same simple (source, sink) flows now exist in both malicious and benign apps.

To combat this problem, we propose an approach that uses a more comprehensive information flow analysis technique. The uniqueness of our approach comes from the following two features. First, our information flow analysis represent an information flow not as a simple (source, sink) pair, but as *a sequence of API calls* from a source to a sink. This gives us the ability to distinguish different flows with same sources and sinks based on the *computation* performed along the information flow. Second, our information flow analysis detects Multi-Flows, flows that either start with a single source and flow to multiple sinks, or start with multiple sources and flow to a single sink. We treat such flows as a single flow, instead of multiple distinct flows. This allows us to examine the *structure* of the flows themselves. We then leverage machine learning techniques to extract features from Multi-Flows and their API sequences (N-gram analysis [2]) and use these features to perform SVM-based classification.

The contributions of this paper are as follows:

- We present Multi-Flow, a new strategy to reveal how an app leverages device sensitive data. We examine patterns of API usage along control flow paths within a Multi-Flow to characterize app behavior.
- An open source implementation of Multi-Flow analysis

Source	Sink
TelephonyManager:getDeviceId	HttpClient:execute
TelephonyManager:getSubscriberId	HttpClient:execute
LocationManager:getLastKnownLocation	Log:d
TelephonyManager:getCellLocation	Log:d

TABLE I
APP INFORMATION FLOWS

and API sequencing in the BlueSeal framework, along with N-gram analysis and a SVM-based classifier.

- A detailed evaluation study, highlighting the differences in old and new apps. We leverage the app behavior extracted as features from both Multi-Flows and their API usage patterns and apply machine learning techniques to automatically identify malware based on the structure of its computation over sensitive data. We test our tool on a set of 1,576 benign apps downloaded from Google Play and 2,422 known malicious apps. Our tool can achieve (84.1%) true positive rate and (90.4%) true negative rate with a false positive rate of (9.6%) when classifying modern malware. Our experiment results show that app behavior difference on device sensitive data can be a significant factor in malware detection.

The rest of the paper is organized as follows: we first present a series of motivating examples in Section II. We discuss Multi-Flow and N-gram analysis of API usage in Section III. Our system design and implementation are discussed in Section IV. We show the effectiveness of our tool in Section V. Related work and conclusions are given in Section VII and Section VIII respectively.

II. MOTIVATION

To illustrate how modern benign and malicious apps can confound detection of malware, consider two apps that contain the same (source, sink) flows shown in Table I. The benign app (*com.kakapo.bingo.apk*), is a popular bingo app available in Google Play and the malicious app claims to be a video player. The malicious app starts a background service to send out premium messages when the app starts and also steals phone info including *IMEI, IMSI*. Notice, that both benign and malicious apps send out `phone identifiers (IMEI, IMSI)` over the Internet and write `location` data into log files. Many previous approaches would consider sending of phone identifiers as an indication of malicious intent [46]. However, we have noticed that sending this information is sometimes used by benign apps, usually as a secondary authentication token for banking apps, or in the case of our bingo app, as a way to uniquely identify a user. In general, it has become more common that benign apps require additional information to provide in-app functionality. Many ad engines collect this kind of information as well [30]. Thus, it is difficult to tell which apps are benign and which are malicious by examining source and sink pairs alone. More information is required to differentiate these two apps.

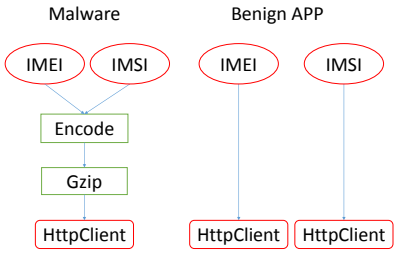

Fig. 1. App Behavior Comparison in Benign and Malware Apps

Let us examine how both our example apps access sensitive data, to see if we cannot differentiate between them. We present the bingo app and the malicious app in the form of decompiled DEX bytecode in code snippets List 1 and List 2, respectively. We observe that the benign bingo app accesses the sensitive data it requires in lines 6, and 12, whereas the malicious app collects the sensitive data in aggregate in a single method in lines 3-4. The malicious app also bundles the data in lines 5-8 and sends the aggregated data over the network in line 10. In contrast, our bingo app does not send data immediately after collecting it. As shown in this example, the two apps contain the same information flows, but the structure of these flows is quite different.

```
1  public static String getLmMobUID(Context context){
2    ...
3    TelephonyManager tm= (TelephonyManager)
4        context.getSystemService("phone");
5    if (isPermission(context,
         "android.permission.READ_PHONE_STATE"))
6      localStringBuffer.append(tm.getDeviceId());
7    ..
8  }
9  public static String getImsi(Context context){
10   TelephonyManager tm = (TelephonyManager)
11       context.getSystemService("phone");
12   param = tm.getSubscriberId();
13   ...
14 }
```

Listing 1. Data Access Code Snippet in Benign App

```
1  private void execTask(){
2    ...
3    this.imei = localObject2.getDeviceId();
4    this.imsi = localObject2.getSubscriberId();
5    str2 = "http://" + Base64.encodebook(
6    "2maodb3ialke8mdeme3gkos9g1icaofm", 6, 3) +
7    "/mm.do?imei=" + this.imei;
8    localStr2 = str2 + "&imsi=" + this.imsi;
9    ...
10   paramString1 =
         ((HttpClient)localObject).execute(localStr2);
11   ...
12 }
```

Listing 2. Data Access Code Snippet in Malware App

The difference becomes even more profound if we examine the computation the apps perform along the code path of

the information flow. Figure 1 shows the information flow view of these two apps. This kind of information can be captured by examining the API call sequence of both programs. This idea is inspired by previous studies on program behavior analysis via sequences of system calls [21, 15]. To illustrate the utility of the approach, we examine the API call sequence results of the two apps along the flow *TelephonyManager:getSubscriberId→HttpClient:execute*. The example API call sequences extracted from both apps are shown in List 4 and List 3 respectively. The lines in black show the same behavior of the two apps, with both preparing to fetch the `IMSI`. The difference between the apps is highlighted in red. The malicious app fetches another phone identifier(`IMEI`) (line 3) right after fetching `IMSI`, then couples this data (line 5) and compresses it (line 6). The benign app, on the other hand, simply checks and uses the network (lines 3-5). By comparing the API sequences we can infer that even though these two apps share the same information flow they differ in app behavior. We can leverage machine learning techniques to discover which behaviors along information flows and Multi-Flows are indicative of malicious code.

```
1  <Context: getSystemService(String)>
2  <TelephonyManager: getSubscriberId()>
3  <TelephonyManager: getDeviceId()>
4  <BasicNameValuePair: <init>(String,String)>
5  <URLEncodedUtils: format(List,String)>
6  <XmlServerConnector: byte[] zip(byte[])>
7  <HttpGet: void <init>(String)>
8  <DefaultHttpClient: void <init>()>
9  <HttpClient: getParams()>
10 <HttpParams: setParameter(String,Object)>
11 <HttpClient: getParams()>
12 <HttpParams: setParameter(String,Object)>
13 <HttpClient: execute(HttpUriRequest)>
```

Listing 3. API Call Sequence in Malware App

```
1  <Context: getSystemService(String)>
2  <TelephonyManager: getSubscriberId()>
3  <PackageManager: checkPermission(String,String)>
4  <WifiManager: getConnectionInfo()>
5  <WifiInfo: getMacAddress()>
6  <TextUtils: isEmpty(CharSequence)>
7  <TextUtils: isEmpty(CharSequence)>
8  <TextUtils: isEmpty(CharSequence)>
9  <HttpGet: <init>(String)>
10 <BasicHttpParams: <init>()>
11 <HttpConnectionParams:
       setConnectionTimeout(HttpParams,int)>
12 <HttpConnectionParams:
       setSoTimeout(HttpParams,int)>
13 <DefaultHttpClient: <init>(HttpParams)>
14 <HttpClient: execute(HttpUriRequest)>
```

Listing 4. API Call Sequence in Benign app

III. Multi-Flow and API Sequence Analysis

A. Multi-Flows

By examining malicious app examples and benign app examples, we observed that a single piece of sensitive resource data may involve in multiple data flows in a malicious app, which means multiple operations are performed on the same piece of data. We propose the concept of Multi-Flow, a new scheme to present the usage of mobile sensitive resources in an app. A Multi-Flow adhere to one of the following two definitions:

1) A single source data flows into multiple sinks.
2) Multiple sources flow into the same sink.

Multi-Flow is one way to describe app behavior on device sensitive data. By analyzing Multi-Flow information we can have a better understanding of sensitive data usage inside an app. For example, in the malicious example mentioned in Section II, we can easily tell that multiple device identifiers are collected at once and sent out over the network. More importantly, the Multi-Flow indicates that the data is sent out not only over the same sink, but also over the same control flow path.

B. App Behavior Analysis via API Sequences

Although Multi-Flow can reveal the usage of sensitive data and the structure of the information flows, a Multi-flow by definition only focuses on discovering the starting point and the end point of data flows, be they simple or complex. To fully understand how an app leverages device sensitive data, analysis of the computation the app performs on the sensitive data is required. One way to extract this kind of information from an app is to discover interactions between an app and platform framework. Even though an app can define tons of operations to manipulate a piece of data, eventually the app must send/leverage this data via platform framework to complete communication with outside world. For example, if an app wants to send out the `DeviceId` over network, it has to leverage the public network APIs of the platform framework to complete this operation. Or if the app wants to write it via logging system, the app must invoke the APIs of `android.uti.Log` package provided by Android framework. Even if the app does nothing but displays sensitive information on the screen, it must do so through the framework GUI APIs. In order to extract the computation performed over sensitive data, we extract the framework API call sequences present in an app. Since we are interested in app behavior related to the usage of device sensitive data, we only analyze the API call sequences present within Multi-Flows instead of all API call sequences contained in the program.

We use the app behavior information extracted from the API call sequences within Multi-Flows to distinguish benign and malicious apps. To do this, we leverage machine learning techniques. However, it is ineffective to use the API call sequences as features directly. This is because the size of API call sequences along with Multi-Flows varies and in most cases, the call sequences of different apps will differ greatly. N-gram technique has been used for malware analysis before and previous studies have proven its effectiveness [20]. In this paper, we leverage this technique to generate features from API call sequences within Multi-Flows and use these features for classification. The details are presented in the next section.

IV. System Desgin

We have built an automated classification system that classifies apps as malicious or benign via analyzing the behavior described in Sections II and III. This classification system is integrated into BlueSeal [38], a static data flow analysis engine originally developed to extract information flows for Android apps. It takes as input the Dalvik Executable (DEX) bytecode for an app, bypassing the need for an app's source. The analysis engine, BlueSeal, is built on top of the Soot Java Optimization Framework [41]. Our design extends BlueSeal to discover Multi-Flows in addition to its native capability to detect simple information flows, a flow from one source to one sink. Our automated classification system performs the following four analysis phases to generate features and perform classification of apps as malicious or benign: (1) Multi-Flow discovery, (2) Control-flow path and call sequence extraction, (3) N-gram feature generation, and (4) Classification. Details for each phase are discussed in the following subsections. Our tool is open-source and available online. Please refer http://blueseal.cse.buffalo.edu/ for details.

A. Multi-Flow Discovery

Traditional information flow analysis mainly focuses on the discovery of a flow from a single source to a single sink. Multi-Flow analysis, on the other hand, examines complex information flows. We have extended BlueSeal to extract these complex information flows, or Multi-Flows, where individual single source to a single sink flows are aggregated and connected. We then analyze these Multi-Flows to classify app behavior on sensitive device data.

The goal of the Multi-Flow detection algorithm is to: (1) create a global graph of complete data flow paths for an app, and (2) detect the intersection between individual data flow paths that represent complex flows. Here, the intersection of two data flow paths simply means two data flow paths share at least one node in the global graph. Thus, if two data flow paths share at least one graph node, then these two data flows belong to the same Multi-Flow.

The Multi-Flow detection algorithm itself works by taking as input BlueSeal's natively detected individual data flow paths, which track information flows with a single source and single sink. We note that this path information does not include the details of the data flow paths. It only detects data sources and sinks, and whether or not any data path exists between them. In other words, BlueSeal does not analyze actual API calls between sources and sinks nor combine data flows to detect Multi-Flows. Therefore, we augment BlueSeal as follows:

- We start the data flow tracking process whenever we encounter a statement containing sensitive API invocation in the program; that is when a device's sensitive data is accessed. This sensitive data access invocation is added as a node in the global graph and is considered the starting point of a data flow path.
- Next, we check each program statement to see if there is a data flow from the current statement to the initial, detected statement. If so, we build an intermediate source node in the global data flow graph, adding an edge from its direct, previous node to it. This step is recursive and if there is a data flow from another program state to the intermediate source node, we create a new intermediate source node as above. These intermediate nodes are critical as they connect together single flows to create Multi-Flows.
- The data flow's path ends when we find a sink point. These three points are able to capture the whole data flow path for a simple information flow while simultaneously outputting a global graph that includes all, potentially interconnected, data flow paths.
- Multi-Flows are detected by iterating through this global graph, finding data flows that may be simple flows or complex flows. The Multi-Flow object maintains the entire data flow path for each source-to-sink pair.
- Lastly, we analyze the control-flow path by examining the data flow paths of each Multi-Flow and extract API call sequences for each, discrete control-flow path.

B. Contro-Flow Graph and API Sequence Analysis

```
1  private void PhoneInfo(){
2    imei = Object2.getDeviceId();
3    mobile = Object2.getLine1Number();
4    imsi = Object2.getSubscriberId();
5    iccid = lObject2.getSimSerialNumber();
6    url = "http://"+str1+".xml?sim="+imei+
7        "&tel="+mobile+"&imsi="+imsi+"&iccid="+iccid;
8    Object2 = getStringByURL(Object2);
9    if ((Object2 != null) && (!"".equals(Object2))){
10     sendSMS(this.destMobile, "imei:" + this.imei);
11   }else{
12     writeRecordLog(url);
13   }
14 }
15 private void sendSMS(String str1, String str2){
16   SmsManager.getDefault().sendTextMessage(str1,
         null, str2,null,null 0));
17 }
18 private void writeRecordLog(String param){
19   Log.i("phoneinfo", param);
20 }
21 public String getStringByURL(String paramString){
22     HttpURLConnection conn =
           (HttpURLConnection)new
           URL(paramString).openConnection();
23     conn.setDoInput(true);
24     conn.connect();
25     return null;
26 }
```

Listing 5. API Call Sequecence Extraction Example

We illustrate this process with an example. List 5 is code extracted from a known malicious app. For simplicity, we remove code not pertinent to the control flow discussion. The general code's data flow structure is shown in Figure 2 and the corresponding control-flow graph is shown in Figure 3.

We then analyze each control flow path statement by statement. We need to perform this step because the Multi-Flow paths do not represent discrete, program execution paths.

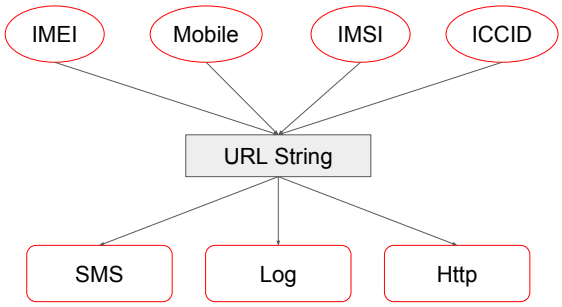

Fig. 2. Data Flow Structure of Example Code Snippet

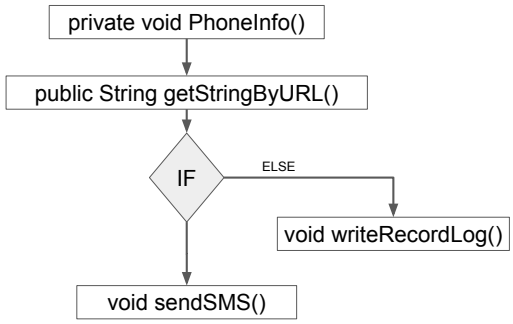

Fig. 3. Control Flow Structure of Example Code Snippet

As our goal is to create a picture of the app's behavior, Multi-Flows must be re-cast as separate execution paths. The example in List 5 contains two execution paths that must be extracted from the larger, singular Multi-Flow structure. Based on path enumeration, we output one API call sequence for each single path. The final output of the example code snippet is shown in Table II. We note that the information flow paths are only a small subset of the program itself, thus we avoid enumerating all paths in the program.

C. N-grams Feature Generation

Next our system generate features for classification purposes using the API call sequences extracted in the previous step. We use the N-grams technique to generate features as N-grams, useful in the detection of malicious codes. Traditionally, the N-grams technique uses byte sequences as input. In our approach, we generate N-grams using API call sequences as input to reveal app behavior. We consider each gram to be a sub-sequence of a given API call sequence. Sequence N-grams are overlapping substrings, collected in a sliding-window fashion where the windows of a fixed size slides one API call at a time. Sequence N-grams not only capture the statistics of sub-sequences of API calls of length `n` but implicitly represent frequencies of longer call sequences as well. An simple example of an API sequence and its corresponding N-grams is shown in Table. III.

Sequence 0	TelephonyManager:getDeviceId() TelephonyManager:getLine1Number() TelephonyManager:getSubscriberId() TelephonyManager:getSimSerialNumber() java.net.URL:openConnection() HttpURLConnection:setDoInput() HttpURLConnection:connect() SmsManager:getDefault() SmsManager:sendTextMessage()
Sequence 1	TelephonyManager:getDeviceId() TelephonyManager:getLine1Number() TelephonyManager:getSubscriberId() TelephonyManager:getSimSerialNumber() java.net.URL:openConnection() HttpURLConnection:setDoInput() HttpURLConnection:connect() Log: int i()

TABLE II
FINAL API CALL SEQUENCE OUTPUT

API Sequence	TelephonyManager:getDeviceId() TelephonyManager:getLine1Number() TelephonyManager:getSubscriberId()
2-grams	TelephonyManager:getDeviceId() TelephonyManager:getLine1Number()
	TelephonyManager:getLine1Number() TelephonyManager:getSubscriberId()

TABLE III
EXAMPLE OF API SEQUENCE AND ITS 2-GRAMS

D. Classification

The last step of our malware classification tool is leveraging machine learning techniques based on N-grams to perform classification and identify significant, different behavior between malicious and benign apps. We generate N-grams for each app and then use every N-gram in any app as a feature to form a global feature space. Based on this global feature space, we generate a feature vector for each app, taking the count of each gram feature into consideration. For example, if a gram feature appears three times in an app, the corresponding value of this gram feature in app's feature vector will be three. Finally, we feed app feature vectors into the classifier. We use *two-class SVM classification* to determine whether an app is malicious or benign. The SVM model is a popular supervised learning model for classification and also leveraged by other systems to perform malicious app detection [7].

V. EVALUATION

Correct selection of training data in classification is very important. There are cases where classifiers work extremely well on one set of data but fail on other sets of data due to over-training [34]. In order to fully evaluate our system and avoid the over-training pitfall, we construct five different sets of apps as follows to better evaluate our tool against different benign and malicious app set combinations. Two sets consist of benign apps, and three sets consist of malicious apps. All our datasets and scripts for evaluation are available online. Please visit http://blueseal.cse.buffalo.edu/ for details.

Benign apps. The benign apps are downloaded from Google Play. This includes two sub-sets. One contains the top 100 most popular free apps across multiple categories from Jan, 2014 and the other contains the top 100 most popular free apps across multiple categories from Aug, 2016. Of those downloaded, we have used 1,576 apps (around 800 apps from each set) that contain information flows for our evaluation. Other apps either have no flow reported by our tool or exceed the execution time limit set for processing the app. This execution time limit is needed because some apps take hours to finish while more than 90% of the apps can be analyzed in well under an hour.

Malicious apps. We use three malware data sets. The first set is from the MalGenome project [46]. Since previous studies use this data set, we leverage it as a comparison point and to to aid in reproducibility. The other malicious apps are from a dataset of over 70,000 malware samples obtained from security operations over a month by a threat intelligence company operating in the United States and Europe. Due to a strict security policy, we cannot disclose further information about this data set. Each app from the 70K set has been scanned through multiple popular anti-virus tools. Meta data is associated with each app including scan result of each anti-virus tool, time discovered, description of the app and so on. Out of the entire set, we have randomly selected 2,422 apps that contain information flows. We divide these malicious apps into two subsets to show our approach works on different sets of malicious apps. Analogous to the benign apps, MalGenome apps represent older, outdated apps while the remaining two sets represent new, recent malware apps.

We label each of the five datasets as follows:

- *Play_2014*: Apps collected from Google Play in Jan, 2014.
- *Play_2016*: Apps collected from Google Play in Aug, 2016.
- *MalGenome*: Malware apps collected from MalGenome project.
- *Malware_1*: First set of malware apps collected from intelligence company.
- *Malware_2*: Second set of malware apps collected from intelligence company.

A. Evaluation Methodology and Metrics

We have used different combinations of these five sets in our experiments to evaluate our classification system. The evaluation process is as follows:

- We use the `10-fold` cross-validation technique to divide apps into a training set and a testing set. We trained the classifier on the feature vectors from a random 90% of both benign and malicious apps. The remaining 10% form the testing dataset. This is a commonly used statistical analysis technique.
- The training set is based on both benign and malicious apps. N-grams generated from these apps are used to form the global feature space. For each app, a feature vector is built based on N-gram features.
- Then feature vectors of apps of the training set are used to train a two-class SVM classifier.
- Lastly, after training, we use the testing set of mixed benign and malicious apps for classification. The classifier then provides a decision on an app, based on its N-grams feature vector, as either "malicious" or "benign".

Upon completion, we collect statistics based on the classification results. We use the following four metrics for our evaluation:

TP True positive rate—the rate of benign apps recognized correctly as benign.

TN True negative rate—the rate of malware recognized correctly as malicious.

FP False positive rate—the rate of malware recognized incorrectly as benign.

FN False negative rate—the rate of benign apps recognized incorrectly as malicious.

The rest of this section details the results.

B. Gram Analysis on Benign and Malicious Apps

In order to illustrate the effects of app grams, we pick one benign app set(*Play_2014*) and one malware app set(*Malware_1*) to compare gram features. The result of our gram analysis on benign apps and malicious apps shows that benign and malicious apps exhibit behavioral differences when analyzed with grams. The details are shown in Table IV, where we list how many unique grams we detect from our data sets, and how many identical grams appear in both benign and malicious apps (shown in the *Overlap* column). In addition, we also examine the gram distribution among apps as shown in Figure. 4. In this figure, we only show the distribution difference of benign and malware apps on 1-gram and 5-gram. The difference of other gram sizes is bounded by these two results. We make the following two observations. First, the total gram counts of benign apps are much greater than that of malicious apps. However, the distribution of grams in benign and malicious apps are quite similar. This mean that malicious apps share more common grams with each other than benign apps.

Second, the ratio of common grams in benign and malicious apps drops significantly as we increase the gram size. These two observations indicate that benign app behavior is diverse across apps while malicious apps may share more common behavior with each other. The first row in the table shows that a significant portion of all API calls in malicious apps appears in benign apps. However, this is no longer true as we increase the gram size, telling us that though benign apps and malicious apps may leverage the same set of the API calls (as captured by the gram size of 1), the usage of these API calls may be quite different (as captured by other gram sizes). Different sizes of grams are leveraged to reveal common behavior.

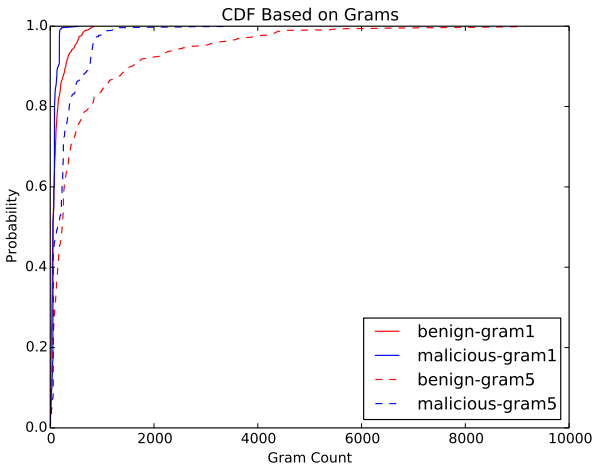

Fig. 4. App Gram Distribution Statistics

Size	Gram Count Benign	Gram Count Malware	Overlap
1	4,080	1,633	1,419
2	33,787	13,027	4,376
3	52,425	13,816	7,634
4	65,154	24,610	5,401
5	273,978	65,865	11,865

TABLE IV
OVERVIEW OF GRAM FEATURES IN BENIGN AND MALICIOUS APPS

gram size	TP	TN	FP	FN	accuracy
1	0.911	0.841	0.159	0.089	0.874
2	1	0.659	0.341	0	0.82
3	1	0.545	0.455	0	0.76
4	0.514	0.967	0.033	0.486	0.762
5	0.474	0.92	0.08	0.526	0.712
1,2	0.81	0.92	0.08	0.19	0.868
1,2,3	0.797	0.943	0.057	0.203	0.874
1,2,3,4	0.772	0.943	0.057	0.228	0.862
1,2,3,4,5	0.646	0.943	0.057	0.354	0.802

TABLE V
GRAM BASED CLASSIFICATION RESULTS OF PLAY_2014 AND MALGENOME APPS

gram size	TP	TN	FP	FN	accuracy
1	0.987	0.71	0.29	0.013	0.796
2	0.95	0.848	0.152	0.05	0.882
3	0.59	0.924	0.076	0.41	0.809
4	0.478	0.939	0.061	0.522	0.765
5	0.333	0.953	0.047	0.667	0.726
1,2	0.961	0.746	0.254	0.039	0.812
1,2,3	0.895	0.852	0.148	0.105	0.865
1,2,3,4	0.632	0.905	0.095	0.368	0.82
1,2,3,4,5	0.587	0.865	0.135	0.413	0.729

TABLE VI
GRAM BASED CLASSIFICATION RESULTS OF PLAY_2014 AND MALWARE_1 APPS

C. *Play_2014 Apps versus MalGenome Apps*

We first show our results with the older benign apps (*Play_2014*) and the older malicious apps (MalGenome). Table V shows that when the gram size is small, it is sufficient to differentiate the MalGenome apps from the benign apps. For the gram size of 1, we achieve a true positive rate of 91.1% and the true negative rate of 84.1%. From this, we can conclude that the usage of single API calls between the benign apps and the MalGenome apps is quite different. As we increase the gram size, we gain more precision for classifying malicious apps while losing precision for classifying benign apps. This is expected as benign apps are more diverse than malicious apps. Increasing the gram size causes a lose of common behavior pattern in benign apps. Based on these observations, we conclude that MalGenome apps are less complicated than benign apps, since much fewer API combinations are detected in these apps(813 1-grams and 13,056 5-grams). Another conclusion we can make is that MalGenome apps misuse single APIs heavily compared to benign apps. This can be captured from the fact that the gram size of 1 is enough to differentiate these malware apps from benign apps.

We also evaluated classification on combined gram features by aggregating different gram size features together as our global feature space. The result is shown as the last 4 rows in Table V. By aggregating different gram features, we can achieve high accuracy rates in classifying both benign apps and malicious apps. We can also see that by aggregating different gram features, we can achieve better precision than using a single gram. However, we also degrade classifier performance by adding too much information. This can be captured by the fact that gram-1,2,3,4,5 has worse precision when compared to other combinations.

D. *Google Play Apps versus Modern Malware Apps*

Next, we show the result with the older benign apps (*Play_2014*) and the newer malicious apps (*Malware_1*). The detailed results are shown in Table VI. Interestingly, our classification with 1-grams does not perform well in distinguishing malicious apps from benign apps. This is quite different from the result with MalGenome apps, which gives us a high precision using 1-grams. However, benign app classification still shows a high precision, since the true positive rate is 98.7%. Our classification on single gram size works best with 2-grams with the true positive rate of 95% and the true negative rate of 84.8%. In this case, we can conclude that recent malicious apps are more similar to benign apps regarding the usage of single APIs than MalGenome apps. However, the computational differences between benign and malicious apps is captured by the fact that we can still achieve very good accuracy in classification using different gram sizes.

The last 4 rows in Table VI show the result or using combined gram sizes. Similar to our previous result with the MalGenome apps, the true negative rate increases along with the increase in gram size, while the true positive rate decreases. The best performance is provided by combining gram sizes of 1, 2, and 3. It has a false positive rate of only 14.8% and a

gram size	TP	TN	FP	FN	accuracy
1	0.921	0.767	0.233	0.079	0.838
2	0.841	0.863	0.137	0.159	0.853
3	0.619	0.863	0.137	0.381	0.75
4	0.475	0.948	0.052	0.525	0.743
5	0.424	0.948	0.052	0.576	0.721
1,2	0.968	0.849	0.151	0.032	0.904
1,2,3	0.857	0.877	0.123	0.143	0.868
1,2,3,4	0.683	0.877	0.123	0.317	0.787
1,2,3,4,5	0.667	0.89	0.11	0.333	0.787

TABLE VII
CLASSIFICATION RESULTS ON PLAY_2016 VS MALWARE_2 APPS

gram size	TP	TN	FP	FN	accuracy
1	0.937	0.795	0.205	0.063	0.86
2	0.937	0.781	0.219	0.063	0.853
3	0.841	0.904	0.096	0.159	0.875
4	0.441	0.883	0.117	0.559	0.691
5	0.39	0.909	0.091	0.61	0.684
1,2	0.937	0.836	0.164	0.063	0.882
1,2,3	0.825	0.89	0.11	0.175	0.86
1,2,3,4	0.703	0.909	0.091	0.297	0.849
1,2,3,4,5	0.688	0.909	0.091	0.313	0.844

TABLE VIII
CLASSIFICATION RESULTS ON PLAY_2016 VS MALWARE_1 APPS

false negative rate of 10.5%. We can also conclude that the aggregated feature space improves the performance more than the single gram size feature space.

Next, we evaluate our approach on different sets of apps. In this experiment, we have used the most recent Google Play apps, labeled as *Play_2016*, as our benign set. We have then chosen a different set of malicious apps, labeled as *Malware_2*. The result is a mixed app set with 753 benign apps and 800 malicious apps for evaluation. The results is shown in Table VII. The results show similar behavior as we increase the gram size. We still can achieve highly precise classification on this new set of apps, while keeping the false positive rates low. Similar to previous results, smaller gram sizes give us better accuracy for both benign apps and malicious apps.

Additionally, we ran our classification on the new Google Play apps versus the other set of malicious apps (*Malware_1*). The evaluation results are shown in Table VIII. As we can see, the results are very similar. These results also support our conclusion above that unlike MalGenome apps, modern malware apps are more similar to benign apps regarding the use of single APIs. However, they are still very different from benign apps from the app behavior perspective. This behavioral difference information can be leveraged to distinguish malicious apps from benign apps. Lastly, these results shows that our approach is effective across all our apps included for evaluation.

E. Simple Information Flow Based Classification

For comparison purpose, we run experiments on simple information flow((source, sink) pair) based classification. The evaluation process are exactly the same as described in Section V-A. The only difference is that we directly use (source, sink) pair as features. We run two experiments over four datasets and show the results in Table. IX. As we see from the table, the simple flow based classification does not perform well on classifying benign and malicious apps in both experiments. This can be captured by the fact of low true negative rate(61.9%) in *Play_2014vsMalware_1* and low true positive rate(58.7%) in *Play_2016vsMalware_2*. This means simple information is insufficient to classify benign and malicious apps and more information is required.

appset	TP	TN	FP	FN	Accuracy
Play_2014vsMalware_1	0.869	0.619	0.381	0.131	0.744
Play_2016vsMalware_2	0.587	0.821	0.178	0.413	0.713

TABLE IX
SIMPLE INFORMATION FLOW BASED CLASSIFICATION RESULTS

F. Comparison to MudFlow

The closest related research to ours is MudFlow [7], which directly leverages information flows as features for classification and leverages machine learning techniques to classify apps in order to detect malware. While MudFlow focuses on classifying apps based on information flows only, we take another step to examine computation is performed along with control flow patch contained within these information flows. As such, MudFlow is a good comparison point for evaluating our techniques. We ran MudFlow on our evaluation datasets, but some of the apps were not successfully processed by MudFlow. The reason for this is either that the MudFlow execution time exceeded the one-hour time limit (which we also use for our tool) or that there was simply no output generated by MudFlow. In such instances we opted to discard these apps from the dataset. Thus, we have used 605 benign apps and 876 malicious apps that are processed correctly by MudFlow. We note that the MalGenome apps are excluded from this comparison as our tests successfully reproduced the results reported in [7]. We would like to thank the authors of MudFlow for providing their tool publicly to facilitate the comparison.

MudFlow provides two different strategies for classification: one-class SVM and two-class SVM. One class SVM is trained only using benign apps, while two-class SVM is trained using both benign and malicious apps. In addition, there are two different settings in two-class SVM. For our evaluation, we have used all three settings.

Table X and Table XI show the results of MudFlow and our approach. As shown, MudFlow can achieve the true negative rate of 69.5% and the true positive rate of 71.5%. In our approach, there is a significant tradeoff between true positive and true negative rates when single gram sizes are used from 1-gram to 5-gram. However, when we combine different gram sizes, we achieve much better accuracy on classification. We get the best performance result when we combine gram sizes of 1, 2, 3, and 4; the true positive rate is 92.2% and the true

run	tnr	tpr	accuracy
one-class	0.678	0.673	0.676
two-class-1	0.665	0.675	0.669
two-class-2	0.695	0.715	0.712

TABLE X
MUDFLOW RESULTS ON EVALUATION APPS

gram size	TP	TN	FP	FN	accuracy
1	0.563	0.889	0.111	0.438	0.735
2	1	0.611	0.339	0	0.794
3	0.969	0.611	0.389	0.031	0.779
4	0.291	0.899	0.101	0.709	0.649
5	0.377	0.89	0.11	0.623	0.657
1,2	0.953	0.708	0.292	0.047	0.824
1,2,3	0.953	0.725	0.275	0.047	0.837
1,2,3,4	0.922	0.75	0.25	0.078	0.831
1,2,3,4,5	0.875	0.722	0.278	0.125	0.788

TABLE XI
GRAM BASED CLASSIFICATION RESULTS ON EVALUATION APPS

negative rate is 75%. We can achieve a better true negative rate (5.5% higher) and a much higher(20%) true positive rate than MudFlow. Recall that MudFlow leverages simple information flows as a feature and our approach leverage N-gram features from Multi-Flow structure for classification. As mentioned in Section II, simple (source, sink) pair cannot distinguish two apps that contain the same flow while app behavior features extracted from Multi-Flow structure can provide us more information to distinguish malicious apps from benign apps. The results clearly prove this point. Our approach collects more information based on app behavior to achieve a better solution, while MudFlow ignores this kind of information.

Due to the number of apps evaluated, the complete effectiveness of both tools is difficult to discern. However, we believe the number of apps evaluated is enough to illustrate they effectiveness of our approach. It is important to note that even though both MudFlow and our approach internally leverage information flows to detect malware, fundamentally we are using different feature sets for classification. We believe these two approaches are complementary to each other.

G. Discussion

Information flows themselves may not provide enough information to distinguish malware apps (missclassified malware from MudFlow). Detailed app behavior, captured by N-grams, is an important feature that can provide critical information used to distinguish malicious apps from benign apps. The detailed app behavior collected by Multi-Flow provides more evidence of the maliciousness of an app(higher true negative rate of our approach). For example, consider the following observation identified by the research. Similar, long API call sequence are less common across benign apps, indicating that benign apps vary greatly in app behavior. However, long API call sequence are common across malware apps and can improve the detection rate of malicious apps, indicating malware shares common behavior patterns. Different sizes of N-grams indicate different complexities of app behavior. Many MalGenome apps can be classified separately from benign apps based on gram-1 features alone, meaning these apps show significant difference of app behavior on single API versus benign apps. In contrast, classification of other more recent malware apps requires more than gram-1 feature. This means these malware are more similar with benign apps than MalGenome ones. However, they can still can be differentiated from benign apps by analyzing detailed app behaviors represented by different gram features.

VI. THREATS TO VALIDITY AND LIMITATIONS

Our classification system leverages static analysis to generate Multi-Flows and thus suffers from the classic limitations of this approach. As new techniques are developed to improve precision of static analysis, specifically static analysis techniques of Android, our tool will be able to leverage these improvements. We currently do not handle analysis of native code in Android apps. Our approach cannot detect malicious behavior that is present in native code. However, we observe that current statistics show that only around 5% of apps make use of native code [47]. Unfortunately, there are a number of know classes of Android malware whose threat vector is primarily located in native code. Our implementation currently does not consider these cases. Other classification schemes use the presence of native code as a feature itself [40] [37].

Another limitation of our approach is that we consider all the apps we downloaded from Google Play Store as benign, but we cannot be completely certain that there are no malicious apps among them. The malicious apps we used in this paper stem from collections of malware where each apps has been identified as malicious at some point. We do not know its main attack type and its malicious code features and our identification scheme is currently agnostic to this information. Lastly, we only consider continuous sub-sequence of API calls in this paper. N-grams features of non continuous sequence of API calls may also be great features for classification purposes.

VII. RELATED WORK

Mobile device security has drawn broad attention among researchers. Various solutions have been proposed to address different security concerns on mobile devices.

A. Information Flow Analysis on Android

TaintDroid [13] is one of the most popular dynamic tools to detect information leaks. By instrumenting an app, TaintDroid can report and stop leaks that occur during execution of the app, but cannot determine if a leak exists prior to execution. Researchers have also developed many static tools to detect information flows: FlowDroid [6], StubDroid [5], CHEX [28], AndroidLeaks [42], SCanDroid [17]. Mann *et al.* created a framework to identify privacy leaks from Android APIs [29], but the framework has not been evaluated on real-world apps. DroidChecker [11] is a static analysis tool aimed at discovering

privilege escalation attacks and thus only analyzes exported interfaces and APIs that are classified as dangerous. ScanDal [26] is an abstract interpretation framework for tracking information flows within apps. Currently, their framework is able to track flows between location information, phone identifiers, camera, and microphone exported to the network and SMS. Yang *et al.* [43] develop a control-flow representation of user-driven callback behavior and propose new context-sensitive analysis of event handlers. Octeau *et al.* [31] presented the COAL language and the associated solver for MVC constant propagation for Android apps. Barros *et al.* [8] present static analysis of implicit control flow for Android apps to resolve Java reflection and Android intents. Slavin *et al.* [39] propose a semi-automated framework for detecting privacy policy violation in Android app code based on API mapping and information flows. Li *et al.* [27] propose `IccTA` to detect privacy leaks among components in Android apps. Yang *et al.* [44] develop a control-flow representation based on user-driven callback behavior for data-flow analyses on Android apps. AppContext [45] extracts context information based on app contents and information flows and differentiates benign and malicious behavior. However, it requires manually labelling of a security-sensitive method call based on existing malware signatures. Huang*et al.* [24] propose a type-based taint analysis to detect privacy leaks in Android apps.

B. Android Malware Detection

There are many general malware detection proposed for Android. Some of these techniques leverage textual information from the app's description to learn what an app should do. For example, CHABADA [18] checks the program to see if the app behaves as advertised. WHYPER [32] leverages NLP techniques to analyze the app description from the market and a semantic model of a permission and determine why an app uses a permission. Meanwhile, AsDroid [23] proposes to detect stealthy malicious behaviors in Android apps by analyzing mismatches between program behavior and user interface. All these techniques rely on either textual information, declared permissions in the manifest file, or on specific API calls, while our approach focuses on analyzing app behaviors based on the app code related to device sensitive data.

Machine learning techniques are also very popular among researchers for detecting malicious Android apps. However, most of these solutions train the classifier only on malware samples and can therefore be very effective to detect other samples of the same family. For example, DREBIN [4] extracts features from a malicious app's manifest and disaassembled code to train their classifier, where as MAST [10] leverages permissions and Android constructs as features to train their classifier. We believe these coarse features are a great mechanisms to filter many apps prior to leveraging techniques like our own, which require more analysis of the app internals. There are many other systems, such as Crowdroid [9], and DroidAPIMiner [3], that leverage machine learning techniques to analyze statistical features for detecting malware. Similarly, researchers developed static and dynamic analyses techniques to detect known malware features. Apposcopy [16] creates app signature by leveraging control-flow and data-flow analysis. RiskRanker [19] performs several risk analyses to rank Android apps as high-, medium-, or low-risk. Sebastian *et al.* [33] analyze dynamic code loading in Android apps to detect malicious behavior. [14], [12] and [20] are all signature-based malware detection techniques and are designed to detect similar malware apps.

C. Other Android Security tools

In addition, researchers have explored many other ways to ensure security on Android. For example, Sadeghi *et al.* [36] develop COVERT to analyze Android app in a compositional manner to detect inter-app and inter-component security vulnerabilities. Jamrozik *et al.* [25] proposed `sandbox mining` to protect device resources from Android apps. The technique first explores software behavior by test generation and extracts the set of resources accessed during tests. This set is then used as a `sandbox`, blocking access to resources not used during testing. However, this approach is highly depending on test generators and may fail to distinguish malicious and benign behavior in the absence of a detailed specification. Our approach does not require a specification after suitable training. Based on our evaluation, only a small number of malicious apps are required for training to yield good results. Rubinov*et al.* [35] develop an approach to protect sensitive data by partitioning app code and handling confidential data in the "secure" world. SUPOR [22] examines UIs to identify sensitive user inputs containing critical user data for security purposes.

VIII. Conclusion

In this paper, we propse a new concept of Multi-Flow to derive app behavior on device sensitive data. We also present an automated classification system that leverages app behavior along with app information flows for classfying benign and malicious Android apps. We have detailed our approach to discover an app's Multi-Flow, extract app behavior features and classification procedure. We also prove the effectiveness of our classification system by presenting evaluation results on Google Play Store apps and known malicious apps. For future work, we plan on refining N-grams feature extraction to eliminate noneffective framework API calls. We also can leverage other machine learning clssification techniques to find the most effective one.

References

[1] Mobile threat report 2016 - mcafee. http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf.

[2] n-gram. https://en.wikipedia.org/wiki/N-gram.

[3] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, pages 86–103, 2013.

[4] D. Arp, M. Spreitzenbarth, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket, 2014.

[5] S. Arzt and E. Bodden. Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 725–735, New York, NY, USA, 2016. ACM.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android appstion in tcb source code. In *PLDI '14*, Edinburgh, UK, 2014.

[7] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 426–436, Piscataway, NJ, USA, 2015. IEEE Press.

[8] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, Lincoln, NE, USA, November 11–13, 2015.

[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[10] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[11] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 125–136, New York, NY, USA, 2012. ACM.

[12] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[14] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[15] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod. Mining control flow graph as api call-grams to detect portable executable malware. In *Proceedings of the Fifth International Conference on Security of Information and Networks*, SIN '12, pages 130–137, New York, NY, USA, 2012. ACM.

[16] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[17] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications.

[18] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035, New York, NY, USA, 2014. ACM.

[19] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[20] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 101–120, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, Aug. 1998.

[22] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 977–992, Berkeley, CA, USA, 2015. USENIX Association.

[23] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[24] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 106–117, New York, NY, USA, 2015. ACM.

[25] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 37–48, New York, NY, USA, 2016. ACM.

[26] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In H. Chen, L. Koved, and D. S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[27] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[28] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[29] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1457–1462, New York, NY, USA, 2012. ACM.

[30] V. Moonsamy, M. Alazab, and L. Batten. Towards an understanding of the impact of advertising on data leaks. *Int. J. Secur. Netw.*, 7(3):181–193, Mar. 2012.

[31] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 77–88, Piscataway, NJ, USA, 2015. IEEE Press.

[32] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 527–542, Washington, D.C., 2013. USENIX.

[33] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic

Code Loading in Android Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[34] F. Provost and T. Fawcett. Robust classification for imprecise environments. *Mach. Learn.*, 42(3):203–231, Mar. 2001.

[35] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 923–934, New York, NY, USA, 2016. ACM.

[36] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using covert. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 725–728, Piscataway, NJ, USA, 2015. IEEE Press.

[37] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim. FLEXDROID: enforcing in-app privilege separation in android. In *NDSS*. The Internet Society, 2016.

[38] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek. Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 515–526, New York, NY, USA, 2014. ACM.

[39] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 25–36, New York, NY, USA, 2016. ACM.

[40] M. Sun and G. Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, pages 165–176, New York, NY, USA, 2014. ACM.

[41] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

[42] M. S. Ware and C. J. Fox. Securing java code: heuristics and an evaluation of static analysis tools. In *Proceedings of the 2008 workshop on Static analysis*, SAW '08, pages 12–21, New York, NY, USA, 2008. ACM.

[43] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.

[44] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.

[45] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 303–313, Piscataway, NJ, USA, 2015. IEEE Press.

[46] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[47] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.