Android Malware Detection using Complex-Flows

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 to detect mobile malware based on information flow analysis. Our approach examines the *structure* of information flows to identify *patterns* of behavior present in them and which flows are *related*, those that share partial computation paths. We call such flows Complex-Flows, as their structure, patterns, and relations accurately capture the complex behavior exhibited by both recent malware and benign applications. N-gram analysis is used to identify unique and common behavioral patterns present in Complex-Flows. The N-gram analysis is performed on sequences of API calls that occur along Complex-Flows' control flow paths. We show the precision of our technique by applying it to four different data sets totaling 8,598 apps. These data sets consist of both recent and older generation benign and malicious apps to demonstrate the effectiveness of our approach across different generations of apps.**

I. Introduction

According to security experts [1], over 37 million malicious applications (apps) have been detected in only a 6-month span in the beginning of 2016. Clearly, malware detection is crucial to combat this high-volume spread of malicious code. Previous approaches for malware detection have shown that analyzing information flows can be an effective method to detect malicious apps [5, 10, 23]. 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. These previous approaches use the presence or absence of certain flows to determine whether or not an app is malicious 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 flow 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. More complex app behavior is involved in leveraging device sensitive data. 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 go 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.

In general, the key to distinguish malicious apps and benign apps is to discover the difference of app behavior on sensitive data usage in apps. We propose a new representation of information flows, called *Complex-Flows*, for a more effective malware detection analysis. Simply put, a Complex-Flow is a set of simple (source, sink) flows that share a common portion of code in a program. For example, a program can read contact information, encrypt it, and store it in storage as well as send it over the Internet. This means that this program has two simple flows—a (contact, storage) flow and a (contact, network) flow—that share a common portion of code in the beginning of each flow (i.e., reading and encryption). Our Complex-Flow then represents both flows together as a set that contains both flows.

Complex-Flows give us the ability to distinguish different flows with same sources and sinks based on the *computation* performed along the information flow as well as the *structure* of the flows themselves. We leverage this insight and develop a new classification mechanism for malware detection that uses Complex-Flows as the basis for classification features. The details of this classification entail an involved discussion, which we defer to Section IV.

In order to evaluate our technique, we have used 3,899 benign apps downloaded from Google Play and 3,899 known modern malicious apps. Our results show that our technique can achieve 97.6% true positive rate and 91.0% true negative rate with a false positive rate of 9.0% when classifying modern malware. This shows that the behavior captured by our Complex-Flows can be a significant factor in malware detection.

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

TABLE I
INFORMATION FLOWS IN BOTH BENIGN AND MALICIOUS APPS

The contributions of this paper are as follows:

- We present Complex-Flow, a new representation to reveal how an app leverages device sensitive data focused on the structure and relationships between information flows.
- We present a new classification mechanism that leverages Complex-Flows to distinguish malicious apps from benign apps.
- We conduct a detailed evaluation study that highlights the differences between historical and recent apps.

The rest of the paper is organized as follows. We first present a series of motivating examples in Section II. We discuss Complex-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 VI and Section VII respectively.

II. MOTIVATION

To illustrate how modern benign and malicious apps can confound malware detectors that leverage information flows, consider one benign and one malicious app 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. The malicious app masquerades as a video player, but it also starts a background service to send out premium messages and steals phone info including *IMEI, IMSI*. Both apps send out `phone identifiers (IMEI, IMSI)` over the Internet and write `location` data into log files. Thus, even if we can detect the information flows shown in Table I we cannot distinguish these two apps.

To combat this problem, many previous approaches would consider sending of phone identifiers as an indication of malicious intent [24]. This approach worked well for some time as this was often considered privileged information. However, we and others have noticed that sending this information is becoming more common in benign apps, usually as a secondary authentication token for banking apps, or in the case of our bingo app and many other games, 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 [19]. 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.

Let us examine how both our example apps access sensitive data, to see if we can differentiate between them. We present the bingo app and the malicious app in the form of decompiled

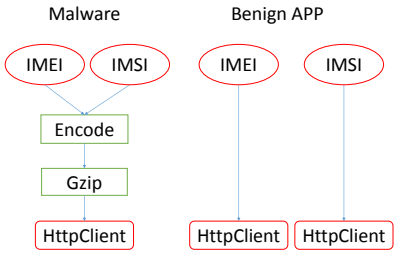

Fig. 1. App Behavior Comparison in Benign and Malware Apps

```
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 }
```

Fig. 2. Data Access Code Snippet in Benign App

DEX bytecode (Android's bytecode format) in code snippets Fig. 2 and Fig. 3, 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.

The difference becomes even more profound if we examine the computation the apps perform along the code path of the information flow. Previous studies [13, 9] have shown that system call sequences effectively capture the computations done in a program; thus, we examine the API call sequences occurring along the flows in both benign and malicious apps, and compare them.

Fig. 1 shows the information flow view of these two apps. In particular, we use the flow *TelephonyManager:getSubscriberId* $\rightarrow$ *HttpClient:execute* as an example to illustrate the differences in benign and malicious apps. Fig. 4 and Fig. 5 show the API call sequences occurring along the flow. 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

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

Fig. 3. Data Access Code Snippet in Malware App

```
<Context: getSystemService(String)>
<TelephonyManager: getSubscriberId()>
<TelephonyManager: getDeviceId()>
<BasicNameValuePair: <init>(String,String)>
<URLEncodedUtils: format(List,String)>
<XmlServerConnector: byte[] zip(byte[])>
<HttpGet: void <init>(String)>
<DefaultHttpClient: void <init>()>
<HttpClient: getParams()>
<HttpParams: setParameter(String,Object)>
<HttpClient: getParams()>
<HttpParams: setParameter(String,Object)>
<HttpClient: execute(HttpUriRequest)>
```

Fig. 4. API Call Sequence in Malware App

3-5).

This example shows that by comparing the API sequences we can infer that even though these two apps share the same information flow they differ in app behavior. Traditional data flow analysis fail to differentiate malicious app behavior from benign one if they both leverage the same set of sensitive data, since it misses the relation of different information flows and the different behavior of these two apps. In our approach, we leverage this insight and represent a set of related simple flows as a Complex-Flow, and develop a machine learning technique to discover which behavior along information flows and Complex-Flows are indicative of malicious code. We further describe this in the next section.

III. Complex-Flows

The analysis of our example apps revealed that it is common for multiple data flows to access sensitive resource data. However, the intent, purpose, and net effect of these operations often differ between the malicious and benign code. We propose the concept of a Complex-Flow, a mechanism that captures the usage of sensitive mobile resources, but also reveals the structure of this usage as well as the relation between different uses.

A. Multi-Flows

To compute Complex-Flows, we must first discover the relationships between simple flows. We call simple flows which are computationally related to one another *Multi-Flows*.

```
<Context: getSystemService(String)>
<TelephonyManager: getSubscriberId()>
<PackageManager: checkPermission(String,String)>
<WifiManager: getConnectionInfo()>
<WifiInfo: getMacAddress()>
<TextUtils: isEmpty(CharSequence)>
<TextUtils: isEmpty(CharSequence)>
<TextUtils: isEmpty(CharSequence)>
<HttpGet: <init>(String)>
<BasicHttpParams: <init>()>
<HttpConnectionParams:
    setConnectionTimeout(HttpParams,int)>
<HttpConnectionParams:
    setSoTimeout(HttpParams,int)>
<DefaultHttpClient: <init>(HttpParams)>
<HttpClient: execute(HttpUriRequest)>
```

Fig. 5. API Call Sequence in Benign app

Abstractly, a Multi-Flow is composed of multiple simple flows, such that any two simple flows in the Multi-Flow share a subset of their computation.

Let SRC be the data source an app accesses. Let SNK be the sink point the data flows into. Let S_n be an intermediate statement in the program where the source data or data derived from the source data is used (i.e. a data flow).

Definition 1: A simple flow, $SRC \rightarrow SNK$, is composed of a sequence of statements $\bar{S}$, which includes SRC and SNK:

$$\bar{S} = SRC \rightsquigarrow S_1 \rightsquigarrow S_2 \ldots \rightsquigarrow S_{n-1} \rightsquigarrow S_n \rightsquigarrow SNK.$$

We say that a sequence $\bar{S}$ is a subsequence of a flow F, written as $\bar{S} \subseteq F$, if $\bar{S}$ is contained within F.

Definition 2: A Multi-Flow represents multiple simple flows that share common computation within a program. Let $\bar{F}$ be a set of all simple flows in a program. A Multi-Flow for a sequence $\bar{S}$, $\bar{F}'(\bar{S})$, is a set of simple flows in $\bar{F}$ that share $\bar{S}$ as a common subsequence. It is defined as:

$$\bar{F}'(\bar{S}) = \{F_i |\ F_i \in \bar{F} \text{ and } \bar{S} \subseteq F_i\}$$

Thus, the simplest Multi-Flow occurs when two simple flows share the same source or the same sink. It is important to distinguish that by source and sink we not only mean a given API call, but where that API occur within the program. Section II provides a real-world Multi-Flow example with multiple device identifiers collected at once and sent out over the network. Here, the data is sent out not only just over the same sink, but also over the same control flow path.

B. Complex Flows

Information flow analysis focuses on discovery of the start and end points of data flows, whether they be simple flows or Multi-Flows. Analysis of the computations captured by Complex-Flows is required to gain understanding of the behavior of the Multi-Flow. Specifically we focus on discovery of the interactions between an app and platform framework. For example, if an app wants to send out the `DeviceId` over network, it must leverage the public network APIs of the platform framework to complete this operation. Or if the app wants to write `DeviceId` via the logging system, it must

invoke the APIs of the Android provided `android.uti.Log` package. Even if the app does nothing but simply display sensitive information on screen, it still must do so through the framework GUI APIs. A formal definition of Complex Flows is as follows:

Definition 3: Let $\bar{S}$ be a simple flow. We define an API sequence of $\bar{S}$ as a filtered sequence over $\bar{S}$ that only contains API call statements. Note that both the source and sink are API calls by definition.

For a formal definition of an API sequence, we write $S \in \overline{API}$, if the statement S is a call to an API function. Then an API sequence of $\bar{S}$ is produced by filtering $\bar{S}$ recursively using the following three rules, which essentially removes all non-API calls from a simple flow (below, S is a single statement, and $\bar{S'}$ is a sequence of statements):
Rule 1: $filter(S \rightsquigarrow \bar{S'}) = S \rightsquigarrow filter(\bar{S'})$ if $S \in \overline{API}$
Rule 2: $filter(S \rightsquigarrow \bar{S'}) = filter(\bar{S'})$ if $S \notin \overline{API}$
Rule 3: $filter(\emptyset) = \emptyset$

Definition 4: We define a Complex Flow CF in terms of a Multi-Flow, $\bar{F}(\bar{S})$ as the set of filtered sequences (i.e., API sequences - $\overline{AS}$) for each flow in the Multi-Flow:

$$CF = \{AS | AS = filter(F), F \in \bar{F}(\bar{S})\}.$$

Definition 5: An N-gram API set is a set of API sequences of size N derived from an API sequence. Formally, a set of N-grams over a filtered sequence is defined as follows, where $|\bar{S'}|$ denotes the size of the filtered sequence $\bar{S'}$:

$$\textit{N-gram}(\bar{S}) = \{\bar{S'} | \bar{S'} \subseteq \bar{S}, |\bar{S'}| = n\}$$

Definition 6: We define all N-grams for a Complex Flow CF as a set of N-gram API sets, one derived from each filtered sequence AS contained in the Complex Flow:

$$\{NG | NG = \textit{N-gram}(AS), AS \in CF)\}.$$

We extract the app's framework API call sequences to capture the computations performed over sensitive data. We only include those sequences present within Complex-Flows. A Complex-Flow, represented as a set of sequences of APIs, including the source and sink pairs of all simple flows present in the Multi-Flow.

IV. System Desgin

We have built an automated malware detection system that classifies apps as malicious or benign via analyzing the N-gram representation of Complex Flows described in Sections II and III. This classification system is integrated into our BlueSeal compiler [20] [14], a static information flow analysis engine originally developed to extract information flows from Android apps. It also can handle information flows triggered by UI events and sensor events. BlueSeal is context sensitive, but is not path sensitive. It takes as input the Dalvik Executable (DEX) bytecode for an app, bypassing the need for an app's source. BlueSeal is built on top of the Soot Java Optimization Framework [21] and leverages both intraprocedural and interprocedural data flow analysis. In addition, BlueSeal is able to resolve different Android specific constructs and reflection. More details are discussed in our previous paper [20].

Our implementation extends BlueSeal to discover Complex-Flows in addition to its native capability to detect simple information flows. The automated classification component performs the following four analysis phases to generate features and perform classification of apps as malicious or benign: (1) Multi-Flow discovery, (2) API 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. We have extended BlueSeal to extract Multi-Flows, where individual single source to a single sink flows are aggregated and connected. We leverage data flow analysis techniques to extract paths contained within each simple flow. If two information flows share a subpath with each other then these two information flows belong to the same Multi-Flow. Each Multi-Flow can contain multiple information flows, which means it can contain multiple sources and multiple sinks. We then analyze these Multi-Flows to extract API sequences present within the Multi-Flow to create Complex-Flows.

The goal of the Multi-Flow detection algorithm is to: (1) create a global graph of complete information flow paths for an app, and (2) detect the intersection between individual information flow paths that represent Multi-Flows. Here, the intersection of two information flow paths simply means two information flow paths share at least one node in the global graph. The Multi-Flow detection algorithm itself works by taking as input BlueSeal's natively detected individual information flow paths, which track simple flows with a single source and single sink. To generate Multi-Flows, we augment BlueSeal as follows:

- Whenever we encounter a statement containing sensitive API invocation (which accesses a device's sensitive data), we add the invocation as a node in the global graph. This 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 the node for the initial statement. This step is recursive and if there is a data flow from another program statement 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 types of points (i.e., source, intermediate, and sink) 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.

```
private void PhoneInfo(){
  imei = Object2.getDeviceId();
  mobile = Object2.getLine1Number();
  imsi = Object2.getSubscriberId();
  iccid = lObject2.getSimSerialNumber();
  url = "http://"+str1+".xml?sim="+imei+
      "&tel="+mobile+"&imsi="+imsi+"&iccid="+iccid;
  Object2 = getStringByURL(Object2);
  if ((Object2 != null) && (!"".equals(Object2))){
    sendSMS(this.destMobile, "imei:" + this.imei);
  }else{
    writeRecordLog(url);
  }
}
private void sendSMS(String str1, String str2){
  SmsManager.getDefault().sendTextMessage(str1,
      null, str2,null,null,0);
}
private void writeRecordLog(String param){
  Log.i("phoneinfo", param);
}
public String getStringByURL(String paramString){
    HttpURLConnection conn =
        (HttpURLConnection)new
        URL(paramString).openConnection();
    conn.setDoInput(true);
    conn.connect();
    return null;
}
```

Fig. 6. API Call Sequence Extraction Example

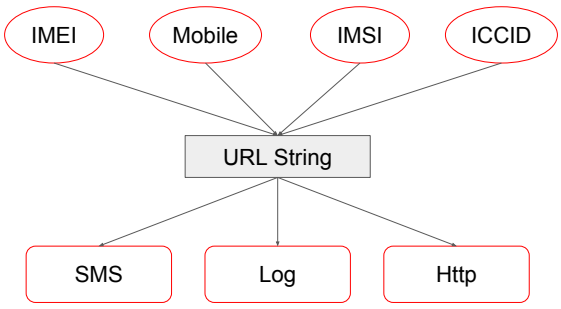

Fig. 7. Data Flow Structure of Example Code Snippet

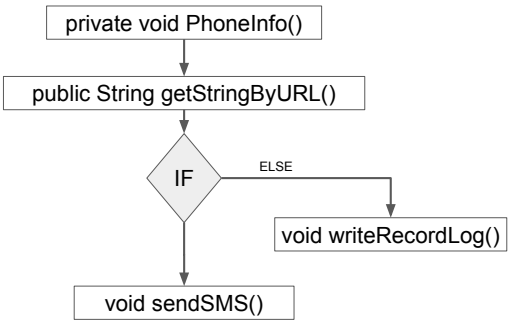

Fig. 8. Control Flow Structure of Example Code Snippet

- Multi-Flows are detected by iterating through this global graph, finding simple data flows as well as Multi-Flows.
- Lastly, we extract API call sequences for all Multi-Flows. While doing so, we analyze control-flow paths in each Multi-Flow to extract precise API call sequences. We discuss this further next.

B. Complex-Flow Extraction with API Sequence Analysis

Although the previous phase gives us the global graph for an app with all Multi-Flows, it does not provide the exact API call sequences occurring along the Multi-Flows, i.e., Complex-Flows. Analyzing Complex-Flows requires us to consider control paths with branches and loops, since they produce separate code paths. For example, if there is an if-else block in-between a source and a sink, there can be two separate API sequences that start with the same source and end with the same sink. Thus, we develop a mechanism to examine all code paths along the Multi-Flows detected by the previous phase, and extract the API call sequences.

Technically, this can be done within the previous phase, as the original BlueSeal implementation already considers control paths when analyzing data flows. However, we implement our API sequence extraction as a separate phase for clean separation of our new logic.

We illustrate this process with an example. Fig. 6 is a code snippet extracted from a known malicious app. For simplicity, we remove other pieces of code not pertinent to our discussion. The general code's data flow structure is shown in Fig. 7 and the corresponding control-flow graph is shown in Fig. 8. Fig. 6 and Fig. 8 show that there are two execution paths that must be extracted from the larger, singular Multi-Flow structure shown in Fig. 7. Thus, we output one API call sequence for each single path. The final output of the example code snippet is shown in Table II.

In order to extract such API sequences, we analyze each control flow path, statement by statement, in the execution order to extract all platform APIs invoked along with Multi-Flows. As mentioned earlier, we consider different branches separately, which means that for each branch point, we create two separate branch paths. For a loop, we consider the execution of its body once if an API is invoked inside a loop. This is due to the fact that precise handling of loops itself is a challenging problem and an active area of research, which requires loop bound analysis followed by unrolling each loop for N times where N is the analyzed bound for the given loop. Previous work proposes a mechanism to precisely handle loops in Android apps [11]; it is our future work to incorporate it. It is worth mentioning here that we have an opportunity to reduce the complexity of precise loop analysis, since our N-gram analysis described next has a maximum bound for an API call sequence, i.e., we are only interested in an API call sequence of size N. This means that we only need to unroll a loop enough times to get an API call sequence of size N, which reduces the complexity of handling loops. However, we leave the full investigation of this as our future work.

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

C. N-gram Feature Generation

Next, our system uses the API call sequences extracted in the previous step to generate features for classification purposes. As mentioned above, the API sequences are the interaction between app and platform, and they represent app behavior regarding sensitive data usage. We use the N-grams technique to generate these features from the API call sequences as N-grams. 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. A simple example of an API sequence and its corresponding N-grams is shown in Table. III. In detail, the first 2-gram indicates that the app access the IMEI and phone number at once; while the second 2-gram indicates that the app access the phone number and IMSI at once.

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 analyzed 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 [5].

V. EVALUATION

To evaluate our classification system, we have collected both benign and malicious app sets. As mentioned earlier, our system is trained on both benign and malicious apps to identify different behavioral patterns between benign and malicious apps. The detailed description of these data sets and the proccess of our evaluation is discussed below.

Benign apps: The benign apps are free apps downloaded from Google Play and include two sub-sets. One contains the top 100 most popular free apps across multiple categories from January, 2014 and the other contains random free apps across multiple categories from Oct, 2016. We have used 3,899 apps in total from the set of apps downloaded, using only those that contain information flows for our evaluation.

Malicious apps: The 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 non-disclosure agreement this set is not publicly available. Each app from the 70K set has been scanned through multiple popular anti-virus tools. Out of the entire set, we have randomly selected 3,899 apps that contain information flows to match up the number of benign apps.

A. Evaluation Methodology and Metrics

The evaluation process is as follows:

- We use the 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.

gram size	TP	TN	FP	FN	accuracy
1	0.945	0.775	0.225	0.055	0.865
2	0.985	0.736	0.264	0.015	0.872
3	0.988	0.659	0.341	0.012	0.833
4	0.974	0.540	0.460	0.025	0.758
5	0.976	0.528	0.472	0.024	0.768
1,2	0.980	0.865	0.135	0.020	0.926
1,2,3	0.976	0.910	0.090	0.024	0.945
1,2,3,4	0.976	0.757	0.243	0.024	0.874
1,2,3,4,5	0.949	0.716	0.284	0.051	0.840

TABLE IV
GRAM BASED CLASSIFICATION RESULTS OF GOOGLE PLAY AND MALWARE APPS

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.

B. Google Play Apps versus Modern Malware Apps

In order to evaluate the effectiveness of our approach, we run our analysis over a mixed set of both benign and malicious APKs, which contains all 3,899 Google Play apps and 3,899 modern malicious apps. The detailed results are shown in Table IV. Interestingly, our classification with single size grams does not perform well in distinguishing malicious apps from benign apps. Furthermore, we also run classification analysis with combined grams. As shown in the table, this strategy works much better than single gram classification strategy. Our classification on combined grams works best with combination of 1-gram, 2-gram, and 3-gram with the true positive rate of 97.6% and the true negative rate of 91.0%. In this case, we can conclude that the usage of single APIs is not enough to distinguish benign and malicious apps. There might be two reasons for this. First, many modern malicious apps are repackaged apps from legitimate apps; secondly, many modern malicious apps attempt to trick people into installing their apps by delivering desired functionality using benign code. However, the fact that we can still achieve very good accuracy in classification using different gram sizes means that computational differences between benign and malicious apps play a significant role in the data sets.

In general, analyzing all Google play and modern malicious apps via our classification system proves that behavior analysis with Complex-flows and N-grams can achieve a good performance at distinguishing malicious apps from benign apps.

C. Discussion

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 Complex-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. classification of modern malware apps requires more than gram-1 feature. This means these malware are similar with benign apps regarding the usage of single APIs. However, they can still can be differentiated from benign apps by analyzing detailed app behaviors represented by different gram features.

VI. RELATED WORK

A. Information Flow Analysis on Android

TaintDroid [8] 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 [4], CHEX [17]. DroidChecker [7] is a static analysis tool aimed at discovering privilege escalation attacks and thus only analyzes exported interfaces and APIs that are classified as dangerous. Li *et al.* [16] propose `IccTA` to detect privacy leaks among components in Android apps. Yang *et al.* [22] develop a control-flow representation based on user-driven callback behavior for data-flow analyses on Android apps. AppContext [23] 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.

B. Android Malware Detection

There are many general malware detection techniques proposed for Android. Some of these leverage textual information from the app's description to learn what an app should do. For example, CHABADA [12] checks the program to see if the app behaves as advertised. Meanwhile, AsDroid [15] 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 [3] extracts features from a malicious app's manifest and disassembled code to train their classifier, where as There are many other systems, such as Crowdroid [6], and DroidAPIMiner [2], that leverage machine learning techniques to analyze statistical features for detecting malware. Mekky *et al.* [18] leverage N-grams to analyze dynamic behavioral traces of malware.

VII. Conclusion

In this paper, we proposed a new concept of Complex Flows 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 classifying benign and malicious Android apps. We have detailed our approach to discover Complex Flows in an app, extract app behavior features, and apply a classification procedure. We show 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 classification techniques to find the most effective ones.

Acknowledgement: This work has been supported in part by an NSF CAREER awared, CNS-1350883.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. Clapp: Characterizing loops in android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 687–697, New York, NY, USA, 2015. ACM.

[12] 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.

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

[14] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek. Flow permissions for android. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, 2013.

[15] 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.

[16] 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.

[17] 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.

[18] H. Mekky, A. Mohaisen, and Z. Zhang. Separation of benign and malicious network events for accurate malware family classification. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*, pages 125–133, 2015.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.