This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

Android Malware Detection using Complex-Flows

Feng Shen†, Justin Del Vecchio†, Aziz Mohaisen‡, Steven Y. Ko†, Lukasz Ziarek†

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.

1 Introduction

According to security experts [1], over 37 million malicious applications (apps) have been detected in only a 6-month span with this trend expected to grow. Efficient 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 certain families of malicious apps [7, 17, 55]. 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 present within the program. 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 externally to the program itself. 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 tested against known malicious apps.

In this paper, we show that there is a need to look beyond simple flows to effectively leverage information flow analysis for malware detection. By analyzing recent 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 and a simple 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.

The key to distinguish malicious apps and benign apps is to discover the difference between app behavior and computation over sensitive data. 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, store it, and 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). Complex-Flow represents them 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 and their structure as the basis for defining classification features. The details of this classification entail an involved discussion, which we

†*F. Shen, J. Del Vecchio, S. Y. Ko, and L. Ziarek are with Department of Computer Science and Engineering, University at Buffalo, The State University of New York; {fengshen, jmdv, mohaisen, stevko, lziarek}@buffalo.edu.*
‡ *A. Mohaisen is with the Department of Computer Science at the University of Central Florida, E-mail: mohaisen@cs.ucf.edu.*
An earlier version of this work has appeared in IEEE ICDCS 2017 [46].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

defer to Section 4.

The goal of this paper is to differentiate malware from benign apps by analyzing app behavior along information flows that leverage sensitive data. We analyze and study this behavior in both benign and malware apps. First, we extract Complex-Flow inside apps in order to filter uninteresting app behaviors inside apps. As such, we focus on app behavior related to sensitive data manipulation. Second, we analyze API sequences to extract app behavior features along information flows present within a Complex-Flow. Lastly, we leverage these features via machine learning techniques for classification. Apps are, therefore, classified based on their structure (represented as Complex-Flow) and their behavior along these flows (represented as API sequences).

Behavior is considered as normal if it is more widely exhibited by benign apps than malicious apps. Behavior is considered abnormal if it is more widely used by malicious apps instead of benign. An app is considered malicious if it exhibits abnormal behavior (e.g., apply obfuscation on sensitive data, aggregate different sensitive sources before leakage, or perform abnormal computation on sensitive data). Through supervised learning, we aim to determine which combinations of these behavior features indicate malicious behavior and which indicate benign behavior. Based on this, we leverage a classification system to distinguish malware from benign apps.

To evaluate our technique, we used 3,899 benign apps downloaded from Google Play, 12,000 known modern malicious apps, and the well-known MalGenome dataset. 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.

Contributions. Our contributions are as follows. First, 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. Second, we present a new classification mechanism leveraging Complex-Flows to distinguish malicious from benign apps. Finally, we conduct a detailed evaluation study that highlights the differences between historical and recent apps.

Organization. The rest of the paper is organized as follows. We first present a series of motivating examples in Section 2. We discuss Complex-Flow and N-gram analysis of API usage in Section 3. Our system design and implementation are discussed in Section 4. We show the effectiveness of our tool in Section 5. Related work and conclusions are given in Section 7 and Section 8 respectively.

2 Motivation

To illustrate how both modern benign and malicious apps can confound malware detectors leveraging information flows, consider one benign and one malicious app that contain the same (source, sink) flows shown in Table 1. 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 1 we cannot distinguish these two apps.

TABLE 1
Information Flows in Both Benign and Malicious Apps

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

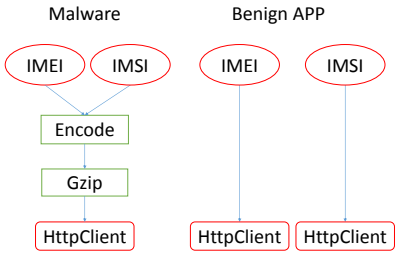


Fig. 1. App Behavior Comparison in Benign and Malware Apps

To combat this problem, many previous approaches would consider sending of phone identifiers as an indication of malicious intent [56]. This approach worked well for some time as this was often considered privileged information. However, we and others [7] [55] 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 [39]. 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 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 [16, 23] 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 *TelephonyMan-*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing


```
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

```
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 }
```

Fig. 3. Data Access Code Snippet in Malware App

ager: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 3-5).

This example shows that by comparing the API sequences we can infer that these two apps differ in behavior even though they share the same information flows. Traditional data flow analysis fails 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. To show how realistic this behavior difference is in real-world apps, we examine both benign and malware test dataset apps leveraging same APIs with encoding and compression as shown in the above example. It turns out that there are 188 malware apps that contain the same behavior while there are only 3 benign apps that exhibit such behavior. 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.

```
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)>
```

Fig. 4. 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)>
```

Fig. 5. API Call Sequence in Benign app

3 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 malicious and benign code. In this section 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.

3.1 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, either by data flow or control flow, *Multi-Flows*. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

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 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 2 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.

3.2 Complex Flows

Information flow analysis focuses on discovering 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: if an app wants to send `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. Below, we provide a formal definition of Complex-Flows.

Definition 3. Let $\bar{S}$ be a simple flow $SRC \rightsquigarrow S_1 \rightsquigarrow ... \rightsquigarrow S_n \rightsquigarrow SNK$, where SRC is a source, SNK is a sink, and S_i is a program statement. 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):

$$\textit{Rule 1}: filter(S \rightsquigarrow \bar{S}') = S \rightsquigarrow filter(\bar{S}') \text{ if } S \in \overline{API} \quad (1)$$

$$\textit{Rule 2}: filter(S \rightsquigarrow \bar{S}') = filter(\bar{S}') \text{ if } S \notin \overline{API} \quad (2)$$

$$\textit{Rule 3}: filter(\emptyset) = \emptyset \quad (3)$$

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.

4 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 2 and 3. This classification system is integrated into our BlueSeal compiler [47] [24], 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 [50] 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 [47] and on the BlueSeal website http://blueseal.cse.buffalo.edu/.

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. Our tool is open-source and available on the BlueSeal website.

4.1 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 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:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing


```
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

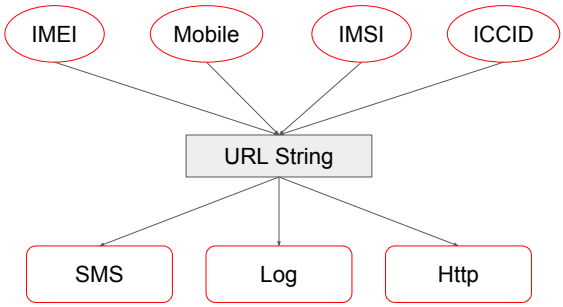


Fig. 7. Data Flow Structure of Example Code Snippet

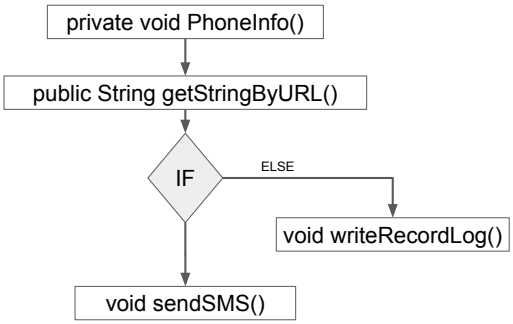


Fig. 8. Control Flow Structure of Example Code Snippet

1. 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.
2. 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.
3. 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.
4. Multi-Flows are detected by iterating through this global graph, finding simple data flows as well as Multi-Flows.
5. We extract API call sequences for all Multi-Flows. While doing so, we analyze control-flow paths in each Multi-Flow to extract API call sequences. We discuss this further next.

As mentioned in [47], we note that our BlueSeal engine will address Android specific constructs. Android Intent will be treated as a sink since it's a potential point to leak data outside. Also, reflection will be resolved if it can be statically determined. Readers can refer to [47] for details.

4.2 Complex-Flow Extraction with API Sequences

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 2.

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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

TABLE 2
Final API Call Sequence Output

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()

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 [18]; 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.

4.3 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. 3. 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.

4.4 Classification

The last step of our malware classification tool is leveraging machine learning techniques on top of N-grams features obtained from the Complex-Flow representation. The goal of this classification to identify significant and different

TABLE 3
Example of API Sequence and its 2-grams

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

behavior between malicious and benign apps. In this process, our system is trained using both benign and malicious apps. Our malware dataset contains malware apps from different families. The practice of assigning a family label to a malware sample is common in the malware detection community. The categorization of apps into families depends on the various rules triggering in (static and dynamic analysis) tools used by the detector; in a way similar to the rules employed by major antivirus scanners incorporated in portals such as VirusTotal [1].

The malicious apps are categorized into different numbers of families based on the different detection tools, and the number of families is on the order of hundreds. Despite the variety of behavior exhibited in the different families, our system is shown to learn how different types of malware leverage information flows and what types of behavior it contains. Besides, our system learns what types of behavior appear when benign apps leverage these information flows. Our system is trained to learn different behavior patterns along with information flows and leverage them as features for classification. In this way, our system will tell if an app leverages information flows in a benign or suspicious way. For a new app, our classification system will compare its behavior pattern, represented as a feature vector obtained from the Complex-Flows representation of the app, and decide whether it is more similar to benign or malicious apps in the training set. In general, our classification system will detect if an information flow is malicious or not based on the app's behavior along the flow.

We note that determining whether an app is malicious or not is rather a complex problem [38], as demonstrated by the multi-billion USD antivirus industry. Furthermore, we note that the initial detection of malicious apps, constituting the ground-truth in our study, falls out of the main scope of this work. For that, the initial labeling of malicious apps follows the industry's best practices, and utilizes dynamic and static analysis (against predefined sets of rules), as well as contexts (users feedback reports) to determine whether an app is malicious or benign. The majority of the apps, however, are determined to be malicious against VirusTotal detection, which utilize a combination of the above static/dynamic techniques as well as permission use and abuse. Thus, the purpose of our approach can be viewed as twofold: (1) a new modality of behavior representation (Complex-Flows) and (2) a new method for label extrapolation based on this modality.

Maliciousness of an app is, therefore, determined by being part of our malware dataset. Our tool tries to match this labeling through extrapolation by classifying an app as malicious through the analysis of the Complex-Flows

1. https://www.virustotal.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

present in the app. In a nutshell, the the app is benign if the Complex-Flows perform meaningful operations similar in structure to other benign apps and malicious if the structure is similar to other malicious apps. Note that our benign dataset is diverse, and covers popular apps, as well as apps from different categories. As those apps do not show a positive detection in public tools, such as VirusTotal, and are not indicated by the Play Store as such, we have assumed them being benign (more details are in section 5.8, when addressing false detection analysis).

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 [7].

5 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 [42]. We use four sets of apps with different characteristics to better evaluate our tool against different benign and malicious app set combinations and avoid the over-training pitfall. Two sets consist of benign apps and two sets consist of malicious apps. All datasets and scripts for evaluation are available. Please visit http://blueseal.cse.buffalo.edu/ for details.

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 all categories (i.e. Art & Design, Beauty, Books & Reference, etc.) 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, excluding the apps that either have no flow reported by our tool or exceed the execution time limit set for processing the app (60 minutes). 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 two malware data sets. The first set is from the MalGenome project [56]. We leverage it as a comparison point and to aid in reproducibility as previous studies rely heavily upon it. 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 non-disclosure agreement this set is not publicly available. Each app from the 70K set has been scanned through multiple popular anti-virus tools. Meta data is associated with each app including scan results from each anti-virus tool, time discovered, description of the app and so on. Out of the entire set, we have randomly selected 3,899 apps that contain information flows to match up the number of benign apps. Of these apps, 27.5% are repackaged apps, meaning they are benign apps with malware injected into them masquerading as benign apps. Analogous to the benign apps, MalGenome apps represent older, outdated apps while the other set represents new, modern malware apps.

We label each of the four datasets as follows:

- *Play_2014*: Apps collected from Google Play in Jan, 2014. The total number of apps is 800.
- *Play_2016*: Apps collected from Google Play in Oct, 2016. The total number of apps is 3,099.
- *MalGenome*: Malware apps collected from MalGenome project. The total number of apps is 800.
- *Malware*: Another set of malware apps collected from intelligence company. The total number of apps is 3,899.

5.1 Evaluation Methodology and Metrics

We have used different combinations of these four 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. Then we rotate on the training and testing dataset. The classification process will be repeated ten times in total and we calculate the results average. 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.

5.2 Runtime Performance

As mentioned earlier, we set an execution time limit on extracting Complex-Flows. Here, we collect statistics on system performance of app execution time of modern malware. All tests are running on machines with twelve Intel(R) Xeon(R) CPU E5645(2.40GHz, 12M Cache). For each app, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

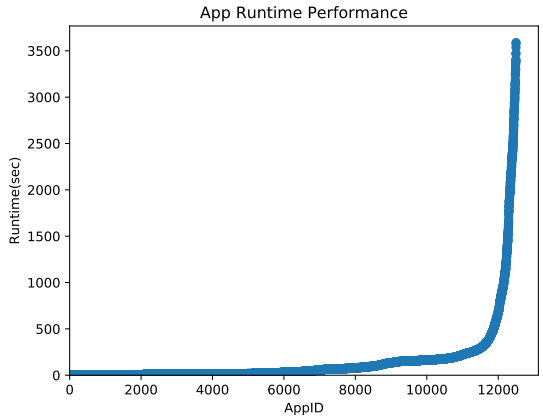


Fig. 9. App Execution Time Performance in seconds.

assign 6G memory to JAVA Virtual Machine. Our system is able to analyze and extract Complex-Flows for 98% of testing apps. There are a few apps (<130 apps) that exceed this execution limit. Among these apps, we increase the time limit to two hours and half of these apps (86 apps) can be analyzed. Only a few apps throw out of memory exception. By increasing JVM memory to 16G and time limit to 24 hours, all apps are able to be analyzed. One of the apps takes around eight hours to finish and all others are able to be done under four hours. Fig. 9 shows the full performance results for apps analyzed under an hour limit. As we can tell from this result, our system can analyze and extract Complex-Flows for 90% of the apps under ten minutes. Only 10% of the apps requires an analysis time greater than that, but can be analyzed in an hour.

5.3 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 4 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 97.5% and the true negative rate of 85.2%.

A manual examination of single API usage in both benign and malware apps shows there are 4,457 distinct APIs in benign apps while there are only 813 for malware apps. The overlap of these two sets is 771. An examination of these 771 shared API calls shows that most of them are sensitive APIs. This indicates that the Malgenome contains malware that is heavily reliant on a specific set of APIs when compared with the benign apps. We also looked at APIs that are exclusive in malware apps, and found them to be primarily APIs related to device WiFi status and database permissions. These APIs are also commonly used in benign apps as well. However, they are not captured along with information flow paths. From this, we can conclude that the usage of single API calls between the benign apps and the MalGenome apps is quite different. By examining the sensitive APIs involved in information flows in these apps, the most common flows indicates that data is often used inside the app for benign apps, while in malicious apps data mainly flows to the network and storage. As we increase the

TABLE 4
Gram Based Classification Results of Play_2014 and MalGenome Apps

gram size	TP	TN	FP	FN	accuracy
1	0.975	0.852	0.148	0.025	0.913
2	0.950	0.699	0.301	0.050	0.822
3	0.980	0.688	0.312	0.020	0.831
4	0.549	0.948	0.052	0.451	0.753
5	0.485	0.952	0.048	0.515	0.725
1,2	0.81	0.92	0.08	0.19	0.868
1,2,3	0.886	0.84	0.16	0.114	0.863
1,2,3,4	0.759	0.889	0.111	0.241	0.825
1,2,3,4,5	0.696	0.938	0.062	0.304	0.819

gram size, we gain more precision for classifying malicious apps while losing precision for classifying benign apps. This is anticipated, since benign apps are more diverse than malicious apps. Increasing the gram size causes a loss of common behavior pattern in benign apps. To this end, we conclude that MalGenome apps are less complicated than benign apps – a result confirmed by our manual inspection. Another conclusion we can make is that MalGenome apps are more interested in a certain set of 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 4. 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 is captured by the fact that gram-1,2,3,4,5 has worse precision when compared to other combinations.

5.4 Google Play Apps versus Modern Malware Apps

In this section, we designed different experiments to evaluate our system thoroughly based on benign apps and modern malware apps. First, we examine the old and new Google Play apps against modern malicious apps individually; then, we run analysis on all benign apps and malicious apps. To do this, we divide modern malicious apps randomly into two subsets to match up with old and new benign apps accordingly and label them as *Malware_1* and *Malware_2*. The detailed results are discussed below.

5.4.1 2014 Google Play Apps vs. Modern Malware Apps

First, 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 5. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

TABLE 5
Gram Based Classification Results of Play_2014 and Malware_1 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	0.967	0.813	0.187	0.033	0.864
1,2,3	0.895	0.852	0.148	0.105	0.865
1,2,3,4	0.624	0.923	0.077	0.376	0.822
1,2,3,4,5	0.596	0.923	0.077	0.404	0.812

TABLE 6
Classification Results on Play_2016 vs Malware_2 Apps

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

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 5 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 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.

5.4.2 2016 Google Play Apps vs. Modern Malware Apps

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 shown in Table 6. 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 7. 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.

TABLE 7
Classification Results on Play_2016 vs Malware_1 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 8
Simple Information Flow Based Classification Results

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

5.4.3 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 subsection 5.1. The only difference is that we use (source, sink) pairs as features instead of examining their structure. We run two experiments over four datasets mentioned in 5.4.1 and 5.4.2 and show the results in Table. 8. 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 is highlighted 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*. The implication is, simple information flows are insufficient to classify benign and malicious apps.

5.4.4 General Google Play Apps vs. Modern Malware Apps

Lastly, 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 9. We have run classification analysis on single size grams as well as combined grams. As shown in the table, the results are quite similar to our previous results. Single size grams do not perform well in distinguishing malicious apps from benign apps, while combined grams work better than single grams. 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%. Again, we conclude that recent malicious apps are more similar to benign apps regarding the usage of single APIs than MalGenome 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 our classification system can still achieve very good accuracy using different gram sizes means that computational differences between benign and malicious apps play a significant role in the data sets.

Due to different complexity of apps we evenly divide both benign and malware APK set into three different categories based on app size, to verify our approach against complexity bias. The big size set contains the top 30% of the APKs based on size; the medium set contains the middle

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

TABLE 9
Gram Based Classification Results of Google Play and Malware Apps

gram size	TP	TN	FP	FN	accuracy
1	0.967	0.788	0.212	0.033	0.858
2	0.961	0.802	0.198	0.039	0.865
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 10
Gram Based Classification Results of Google Play and Malware Apps with Big Size Set

gram size	TP	TN	FP	FN	accuracy
1	0.958	0.685	0.315	0.042	0.806
2	0.92	0.762	0.238	0.08	0.836
3	0.8	0.75	0.25	0.2	0.774
4	0.838	0.797	0.203	0.162	0.818
5	0.701	0.763	0.238	0.299	0.732
1,2	0.972	0.775	0.225	0.028	0.863
1,2,3	0.915	0.820	0.180	0.085	0.863
1,2,3,4	0.831	0.798	0.202	0.169	0.813
1,2,3,4,5	0.817	0.798	0.202	0.183	0.806

TABLE 11
Gram Based Classification Results of Google Play and Malware Apps with Medium Size Set

gram size	TP	TN	FP	FN	accuracy
1	0.762	0.797	0.203	0.238	0.778
2	0.980	0.759	0.241	0.020	0.883
3	0.989	0.798	0.202	0.011	0.894
4	0.969	0.756	0.244	0.031	0.871
5	0.968	0.667	0.333	0.032	0.826
1,2	0.990	0.747	0.253	0.0109	0.883
1,2,3	0.990	0.823	0.177	0.0109	0.917
1,2,3,4	0.980	0.759	0.240	0.020	0.883
1,2,3,4,5	0.989	0.667	0.333	0.011	0.837

TABLE 12
Gram Based Classification Results of Google Play and Malware Apps with Small Size Set

gram size	TP	TN	FP	FN	accuracy
1	0.962	0.864	0.136	0.039	0.920
2	0.976	0.655	0.346	0.024	0.847
3	0.988	0.597	0.404	0.013	0.825
4	0.987	0.542	0.458	0.013	0.794
5	0.985	0.348	0.652	0.015	0.659
1,2	0.987	0.712	0.288	0.013	0.869
1,2,3	0.974	0.729	0.271	0.026	0.869
1,2,3,4	0.974	0.678	0.322	0.026	0.847
1,2,3,4,5	0.962	0.667	0.333	0.039	0.809

30% of the apps; finally, the small set contains the rest of the apps. We also run the classification analysis based on these different sizes. Table 10, Table 11 and Table 12 shows the results for big, medium and small set respectively. As shown in Table 10 and Table 11, the results are similar. The single gram classification does not perform well, but the combined gram classification can achieve high precision. The interesting part is in classification on the small-size sets. As shown in Table 12, this result shows similar behavior with MalGenome apps, i.e., high precision with 1-gram classification. Indeed, these apps are similar in nature to MalGenome apps that were collected over 5 years ago—they are small in size and less complex, and exhibit simple malicious behavior.

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. However, even though the performance of the above results is good, there is one issue that the TN precision varies a lot while the TP precision remains high. We have hypothesized one possible reason for this, which is the imbalanced code size for benign and malicious apps. That is, the code size for benign apps could be much larger than malicious apps, which might affect our results.

To account for this potential size bias, we have designed another experiment that balances the code size of benign and malicious apps instead of balancing the number of apps. This set contains 876 benign apps and 1,352 modern malicious apps, but the total code size of both sets are similar. The detailed results are shown in Table 13. The results show slightly different behavior than our previous results in Table 9, i.e., the precision on malware classification increases with gram size while the precision on benign apps decreases. Nevertheless, our system can still achieve high precision distinguishing malicious apps from benign ones, achieving 96.8% on true positive and 84.9% on true negative.

5.5 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. MudFlow is able to identify new malware as its focus is on identifying abnormal usage. Even though the main goal of MudFlow is to detect abnormal usage of information flows, the authors also provide strategy to identify malware based on its flow of sensitive data. We leverage this feature of MudFlow as a comparison point to differentiate classification of malware based on information flows compared to Complex-Flows. We structure our comparison of malware classification similarly to that made by other researchers [19], who report similar result leveraging MudFlow. Specifically, we evaluate whether Complex-Flows are a better factor than simple information flows in malware classification. We believe that MudFlow and our approach are complementary to each other, allowing MudFlow to identify malware based on both abnormal information flows and Complex-Flows.

We have obtained MudFlow from the authors and run MudFlow on our evaluation datasets. However, 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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

TABLE 13
Classification Results on Google Play vs Malware Apps with Balanced Code Size

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 14
MudFlow Results on Evaluation Apps

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

there are two different settings in two-class SVM. For our evaluation, we have used all three settings.

Table 14 and Table 15 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 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 2, 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 highlight 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. Evaluation results show that app behavior features captured by Complex-Flows can be am effective factor to classify malware from benign apps. 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. Again, we believe these two approaches are complementary to each other.

5.6 Classification Precision and Recall Rates

To understand performance of our classification system on different malicious and benign apps, we summarize experimental results and show how our system performs. Table 16 shows the Precision-Recall rate based on different experiments we mentioned above. Here, we show results from Play_2014 against MalGenome to represent old malicious and benign apps, and results from Play_2016 against

TABLE 15
Gram Based Classification 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 16
Gram Based Classification Precision and Recall.

dataset	gram size	precision	recall
Play_2014vsMalGenome	1	0.868	0.975
Play_2016vsMalware_1	1,2	0.851	0.937
PlayvsMalware_SizeBig	1,2,3	0.836	0.915
PlayvsMalware_SizeMedium	1,2,3	0.848	0.990
PlayvsMalware_SizeSmall	1	0.876	0.962

Malware_1 apps to represent modern malicious and benign apps. Finally, we show results from app size related experiments, which include big, medium and small sized apps. For each of them, we calculate precision and recall statistics based on different gram strategy we leverage on each dataset and present our best system performance here. Gram size column indicates the best gram strategy for each dataset. Plus, we present precision/recall curve to show an overview of our classification system performance for these datasets over all gram strategies, as shown in Figure. 10. We also highlight the best performance gram strategy as indicated with arrows in this figure. This, again, shows that our system can achieve a good performance on classification malware from benign apps by applying different gram strategies on different datasets.

As we can see, old malicious apps are easier to detect since our system can distinguish them from benign apps based on size 1 gram with a high precision and recall rate. While both modern malicious and benign apps are more complicated, single API usage analysis cannot differentiate malware from benign apps. However, different behavior patterns in malicious and benign apps can still be captured when leveraging complex combination of grams. In this dataset, we can achieve a good detection performance by combining 1-gram and 2-gram. Last but not least, modern big and medium size malicious apps are complicated and more difficult to detect than small size ones. This is shown in Table 16 as it requires complex strategy to achieve a good precision and recall rate for big and medium size apps while it is enough to leverage single API usage analysis for small ones.

5.7 Classification based on SVM with Balanced Training

In order to evaluate the efficiency of our classification system on imbalanced training classifier, we design another experiment with more malware apps. In this section, we managed to run another 12,000 malware apps and run our classification system over them against previous 3,899 Play Store apps. The results are shown in Table 17. Similar to other evaluation settings, we have run classification analysis on single size grams as well as combined grams. As shown

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

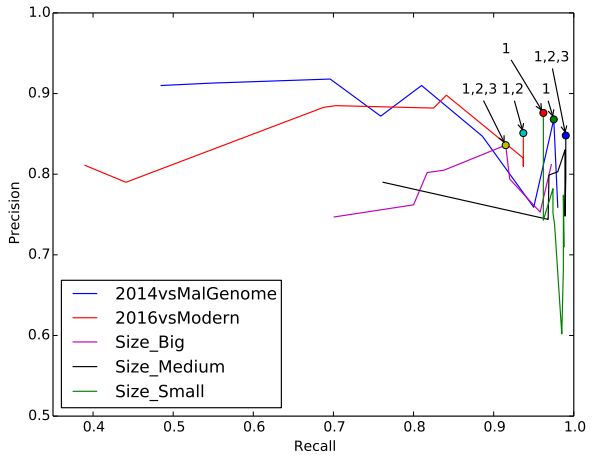


Fig. 10. Precision and Recall Curve.

TABLE 17
Gram Based Classification Results with Balanced Training.

gram size	TP	TN	FP	FN	accuracy
1	0.512	0.908	0.092	0.488	0.794
2	0.623	0.731	0.269	0.377	0.7
3	0.592	0.821	0.179	0.408	0.755
4	0.401	0.851	0.149	0.599	0.722
5	0.320	0.89	0.11	0.680	0.727
1,2	0.891	0.750	0.250	0.109	0.791
1,2,3	0.871	0.892	0.108	0.129	0.886
1,2,3,4	0.7	0.911	0.089	0.3	0.851
1,2,3,4,5	0.571	0.920	0.08	0.429	0.820

in the table, the results, show that single size gram does not perform well in distinguishing malicious apps from benign apps, while combined grams work better than single gams. Our classification on combined grams works best with combination of 1-gram, 2-gram and 3-gram with the true positive rate of 87.1% and the true negative rate of 89.2%. Interestingly, unlike the previous results, the true positive rate never achieves a high precision with single size gram classification. We believe this is caused by the imbalanced number of benign and malware apps. This also tells us that modern malware apps are more similar to benign apps. Even though there is a slightly drop on classification performance, our classification system can still achieve good accuracy using different gram sizes. We can conclude that computational differences between benign and malware apps is a very important factor in distinguishing malicious apps from benign apps.

5.8 Manual Validation

False positives and false negatives are well-known limitations of static analysis and apply to our system as well. Since benign apps are directly downloaded from Play Store, it is possible that this set contains undiscovered malicious apps. We manually validate our system based on our classification results to account for this possibility. We choose fifty benign apps categorized as malicious by our system for manual examination. Most of these apps include multiple third-party libraries such as ad libraries, which likely contribute to the malicious rating. These libraries are complex and lack detailed documentation, making it difficult for us to determine the maliciousness of each and every apps. However, we do find some interesting cases, which we outline below.

Consider, *photo.android.hd.camera.apk*, which claims it is a camera usage app. We found during our manual inspection that this app includes a third party library known as *umeng*, which is known as a high risk adware library. In particular, this library has a capability to download and request installation of new apps. It also monitors running apps on device and sends this process list to a remote location. It also sends device information, such as *IMEI*, location and network info to remote servers. Another interesting case is the benign app *com.necta.aircall_accept.free.apk*, which has over a million downloads. This app is categorized as benign by our classification system. However, based on one of the results of an online malware detection services, it is reported as malicious in our data set. We manually examined the source code of this app to understand this discrepancy. We found it to be a phone call app, which also monitors incoming and outgoing calls on the device. It can also receive and send SMS. Based on our observation and manually analyzed purpose of the app, we do not find any inherent malicious behavior though it is understandable why some tools may classify it as malware based solely on its type of activities.

5.9 Discussion

Information flows themselves may not provide enough information to distinguish malware apps (misclassified 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 ComplexFlow 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 modern malware apps requires more than gram-1 feature. This means these malware are more similar with benign apps than the MalGenome ones. However, they can still can be differentiated from benign apps by analyzing detailed app behaviors represented by different gram features.

6 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

detect malicious behavior that is present in native code. We observe that current statistics show that around 5% to 30% of apps make use of native code [57] [3].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 [49] [45].

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 (§4.4 and §5.8). 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.

7 Related Work

Information Flow Analysis on Android. TaintDroid [14] 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, `e.g.` FlowDroid [6], StubDroid [5], CHEX [34], MutaFlow [36]. However, these tools only detect information flows as single source and sink pairs and report potential leakage. We take a step further to analyze app behavior along with information flows and interaction among different flows. BidText [26] is an static technique to detect sensitive data disclosure by leveraging information flow analysis and type propagation. 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. Instead of analyzing simple API usage, we leverage API sequences as features to define app behavior. Huang *et al.* [28] propose a type-based taint analysis to detect privacy leaks in Android apps. Relda2 [52], a light-weight, scalable and practical static analysis tool, leverages information flow analysis to detect resource leaks in the byte-code of Android apps automatically. However, this tool has been applied to 103 real world apps for testing. We tested our tool on more than 12,000 real world apps for evaluation. Yang *et al.* [53] develop a control-flow representation of user-driven callback behavior and propose new context-sensitive analysis of event handlers. Octeau *et al.* [40] 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. In our tool, we resolve Android specific structs and reflection usage as well. We believe these tools can be complementary to each other. Slavin *et al.* [48] propose a semi-automated framework for detecting privacy policy violation in Android app code based on API mapping and information flows. Li *et al.* [32] propose `IccTA` to detect privacy leaks among components in Android apps. Yang *et al.* [54] develop a control-flow representation based on user-driven callback behavior for data-flow analyses on Android apps. These tools use traditional information flows as feature to detect leakage and pattern based dangerous behavior. However, as we discussed above, both modern benign and malware apps leverage information flows heavily. It requires more information, such as Complex-Flow, to distinguish them. AppContext [55] extracts context information based on app contents and information flows and differentiates benign and malicious behavior. However, it requires manual labelling of security-sensitive method calls based on existing malware signatures.

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 [20] checks the program to see if the app behaves as advertised. Kim *et al.* [30] propose API birthmarks to characterize unique app behaviors, and develop a robust plagiarism detection tool using API birthmarks. Meanwhile, AsDroid [27] 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, 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 disassembled code to train their classifier, where as MAST [10] leverages permissions and Android constructs as features to train their classifier. PFESG [51] merely uses permissions with high utilization, which avoids over-fitting of classification models and quantifies each permission's ability to identify malware. Mahindru *et al.* [35] extract a set of 123 dynamic permissions and evaluate a number of machine learning classification techniques on the newly designed dataset for detecting malicious Android applications. We believe these coarse features are great mechanisms to filter many apps prior to leveraging techniques like our own, which require more analysis of the app internals. McLaughlin *et al.* [37] propose a novel android malware detection system that uses a deep convolutional neural network (CNN) based on static analysis of the raw opcode sequence from a disassembled program. Instead, our system filters out uninterested sequence and focuses on sensitive data related app behaviors instead of using opcode sequence directly. SecureDroid [12] considers different importances of the features to the classification problem, and present a novel feature selection method to make the classifier harder to be evaded. It also proposes an ensemble learning approach by aggregating individual classifiers to improve the system security. Target [33] is a hybrid system featuring both static and dynamic analysis. Its static analysis is based on user permissions, signatures and source code, and dynamic analysis is based on the behavior of running mobile

applications. While we focus on extracting new behavior features from apps, we believe these tools can be complementary to each other. There are many other systems, such as Crowdroid [9], and DroidAPIMiner [2], that leverage machine learning techniques to analyze statistical features for detecting malware. Leeds *et al.* [31] leverage machine learning to differentiate between benign and malicious apps based on permission requests and system calls. However, this has been applied to a small dataset. Our approach focuses on detailed app behaviors from large sets of apps and can achieve a better detection rate. Similarly, researchers developed static and dynamic analyses techniques to detect known malware features. Apposcopy [17] creates app signature by leveraging control-flow and data-flow analysis. RiskRanker [21] performs several risk analyses to rank Android apps as high-, medium-, or low-risk. Sebastian *et al.* [41] analyze dynamic code loading in Android apps to detect malicious behavior. [15], [13] and [22] are all signature-based malware detection techniques and are designed to detect similar malware apps.

Other Android Security Tools. In addition, researchers have explored many other ways to ensure security on Android. These tools leverage techniques on different aspects to protect sensitive data, such as compositional analysis, `sandbox mining`, partitioning app code to handle confidential data or UI examination [44] [29] [43] [25].

8 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 award, 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 *Proc. of SecureComm 2013*, 2013.

[3] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Proc. of NDSS 2016*, 2016.

[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 *Proc. of ICSE 16*, 2016.

[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 *ICSE '15*, Piscataway, NJ, USA, 2015.

[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 '15*, Lincoln, NE, USA, 2015.

[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *SPSM '11*, 2011.

[10] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *Proc. of WiSec '13*, New York, NY, USA, 2013.

[11] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proc. of WiSec '12*, 2012.

[12] L. Chen, S. Hou, and Y. Ye. Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 362–372, New York, NY, USA, 2017. ACM.

[13] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. of SP '05*, Washington, DC, USA, 2005.

[14] 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 *Proc. of OSDI'10*, Berkeley, CA, USA, 2010.

[15] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of CCS '09*, 2009.

[16] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod. Mining control flow graph as api call-grams to detect portable executable malware. In *Proc. of SIN '12*, 2012.

[17] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proc. of FSE '14*, 2014.

[18] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. Clapp: Characterizing loops in android applications. In *Proc. of ESEC/FSE '15*, 2015.

[19] J. Garcia, M. Hammad, and S. Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Trans. Softw. Eng. Methodol.*, 26(3):11:1–11:29, Jan. 2018.

[20] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proc. of ICSE '14*, 2014.

[21] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proc. of MobiSys '12*, 2012.

[22] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Proc. of RAID '09*, 2009.

[23] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 1998.

[24] 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.

[25] 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 *Proc. of SEC'15*, 2015.

[26] J. Huang, X. Zhang, and L. Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.
[27] 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 *Proc. of ICSE '14*, 2014.
[28] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for android. In *Proc. of ISSTA'15*, 2015.
[29] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller. Mining sandboxes. In *Proc. of ICSE 16*, 2016.
[30] D. Kim, A. Gokhale, V. Ganapathy, and A. Srivastava. Detecting plagiarized mobile apps using api birthmarks. *Automated Software Engg.*, 23(4):591–618, Dec. 2016.
[31] M. Leeds, M. Keffeler, and T. Atkison. A comparison of features for android malware detection. In *Proceedings of the SouthEast Conference*, ACM SE '17, pages 63–68, New York, NY, USA, 2017. ACM.
[32] 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 *Proc. of ICSE '15*, 2015.
[33] J. Lin, X. Zhao, and H. Li. Target: Category-based android malware detection revisited. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '17, pages 74:1–74:9, New York, NY, USA, 2017. ACM.
[34] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of CCS '12*, 2012.
[35] A. Mahindru and P. Singh. Dynamic permissions based android malware detection using machine learning techniques. In *Proceedings of the 10th Innovations in Software Engineering Conference*, ISEC '17, pages 202–210, New York, NY, USA, 2017. ACM.
[36] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller. Detecting information flow by mutating input data. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, 2017.
[37] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 301–308, New York, NY, USA, 2017. ACM.
[38] A. Mohaisen and O. Alrawi. Av-meter: An evaluation of antivirus scans and labels. In *Proc. of DIMVA 2014*, 2014.
[39] V. Moonsamy, M. Alazab, and L. Batten. Towards an understanding of the impact of advertising on data leaks. *Int. J. Secur. Netw.*, 7, 2012.
[40] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proc. of ICSE '15*, 2015.
[41] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proc. of NDSS '14*, San Diego, CA, 2014.
[42] F. Provost and T. Fawcett. Robust classification for imprecise environments. *Mach. Learn.*, 42, 2001.
[43] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proc. of ICSE '16*, 2016.
[44] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using covert. In *Proc. of ICSE '15*, 2015.
[45] 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.
[46] F. Shen, J. D. Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek. Android malware detection using complex-flows. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA*, 2017.
[47] 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. ACM, 2014.
[48] 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 *Proc. of ICSE '16*, 2016.
[49] M. Sun and G. Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proc. of WiSec '14*, 2014.
[50] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proc. of CASCON '99*, 1999.
[51] C. Wang and Y. Lan. Pfesg: Permission-based android malware feature extraction algorithm. In *Proceedings of the 2017 VI International Conference on Network, Communication and Computing*, ICNCC 2017, pages 106–109, New York, NY, USA, 2017. ACM.
[52] T. Wu, J. Liu, X. Deng, J. Yan, and J. Zhang. Relda2: An effective static analysis tool for resource leak detection in android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 2016.
[53] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proc. of ICSE '15*, 2015.
[54] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proc. of ICSE '15*, 2015.
[55] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. of ICSE '15*, 2015.
[56] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of SP '12*, 2012.
[57] 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 *Proc. of NDSS 2012*, 2012.

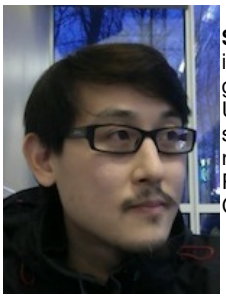
9 BIOGRAHPY

Feng Shen is currently a Ph.D candidate at University at Buffalo, where he focuses on static data flow analysis for Android security. Before joining UB, he graduated with MS from Arkansas State University. Shen is currently leading the BlueSeal team. The BlueSeal team mainly focus on research related to mobile security. The BlueSeal team investigates static analysis and dynamic instrumentation techniques onAndroid. He is co-advised by Steve Ko and Lukas Ziarek. His research interest is in security, distributed systems, programming languages, and mobile computing.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2861405, IEEE Transactions on Mobile Computing

Mr. Del Vecchio is a senior research scientist with CUBRC and has worked as a program manager for U.S. government sponsored research programs for the past 12 years. He currently manages a project that seeks to align and provide analytics for the many, heterogeneous data sources available to Intel analysts. Mr. Del Vecchio is also a Ph.D candidate at the University at Buffalo where he focuses on static code analysis and reverse engineering.

Aziz Mohaisen is an Associate Professor in the Department of Computer Science, with a joint appointment in the Department of Electrical and Computer Engineering, at the University of Central Florida. His research interests are in the areas of systems, security, privacy, and measurements. His research work has been featured in popular media, including MIT Technology Review, the New Scientist, Minnesota Daily, Slashdot, The Verge, Deep Dot Web, and Slate, among others. He obtained his Ph.D. from the University of Minnesota. He is a senior member of IEEE.



Steven Y. Ko Steve Ko is an assistant professor in the Department of Computer Science and Engineering at the University of Buffalo, The State University of New York. His research interest spans in systems and networking, with the current focus on mobile systems. He received his PhD from the University of Illinois at Urbana-Champaign in Computer Science.

Lukasz Ziarek received the B.S. degree in Computer Science from the University of Chicago and the Ph.D. degree in Computer Science from Purdue University. Lukasz Ziarek is an assitant professor in the Department of Computer Science and Engineering at the University of Buffalo, The State Univeristy of New York. His research interests are in programming langauges and real-time systems.