String Analysis of Android Applications

Justin Del Vecchio, Feng Shen, Kenny M. Yee, Boyu Wang, Steven Y. Ko, Lukasz Ziarek
University at Buffalo, The State University of New York
{jmdv, fengshen, kennyyee, bwang24, stevko, lziarek}@buffalo.edu

***Abstract*—The desire to understand mobile applications has resulted in researchers adapting classical static analysis techniques to the mobile domain. Examination of data and control flows in Android apps is now a common practice to classify them. Important to these analyses is a fine-grained examination and understanding of strings, since in Android they are heavily used in intents, URLs, reflection, and content providers. Rigorous analysis of string creation, usage, and value characteristics offers additional information to increase precision of app classification.**

This paper shows that inter-procedural static analysis that specifically targets string construction and usage can be used to reveal valuable insights for classifying Android apps. To this end, we first present case studies to illustrate typical uses of strings in Android apps. We then present the results of our analysis on real-world malicious and benign apps. Our analysis examines how strings are created and used for URL objects, Java reflection, and Android intents, and infers the actual string values used as much as possible. Our results demonstrate that string disambiguation based on creation, usage, and value indeed provides additional information that may be used to improve precision of classifying application behaviors.

I. Introduction

Android is one of the most popular platforms for mobile computing. Countless numbers of devices have been released internationally and Android's official app store, Google Play, contains more than a million apps. However, along with this popularity comes an increase in malicious apps. It was reported in 2013 that 97% of all mobile malware targeted Android [1] and this number has been sharply growing.

This increase in malware has led to the creation of many static analysis tools for identifying malicious Android apps. By identifying malware prior to its release, these tools can offer enhanced security and privacy for end users who download and use Android apps. Many different types of analyses have proven to be useful such as data and control flow analysis [2, 3, 4], call graph analysis [5], and sensitive API analysis [6, 7]. These static analysis approaches provide awareness of potential harms to users.

In this paper, we show that analysis of strings and how they are created in Android apps can be useful in identifying malicious apps. The usefulness of string analysis is derived from two aspects. First, Android apps heavily rely on strings in order to perform potentially sensitive or malicious tasks, such as reading contact information and device IDs, accessing storage, calling hidden APIs, and using remote servers. Second, string analysis is complementary to other types of analysis, and contributes to improving their precision. For example, data and information flow analysis can benefit from additional string analysis if there is a flow from the code that reads the device ID to the code that sends it out to a remote server. By employing additional string analysis, data and information flow analysis will have a chance to differentiate which remote server the flow is intended for, and if it should be considered malicious.

In order to show the usefulness of string analysis, we report our preliminary results in this paper. Our string analysis focuses on analyzing Android-specific uses of strings; these include strings for intents (Android's messaging data structure), Java reflection, and network URLs. Our goal for analyzing these strings is three-fold. First, we aim to verify if the value of a string object indicates its intended use. For example, a URL, a class name used in reflection, or an obfuscated string can indicate the purpose of the code that uses them. Second, we aim to verify if the provenance of a string indicates its intended use. For example, use of strings that come from external sources such as configuration files may be an attempt at subterfuge. Third, we aim to verify if construction of strings provides information on the complexity required in creation of strings, for example a multi-part string for a URL. Analysis of strings in the context of these types of characteristics can provide a wealth of information useful in improving the precision of flow analysis, especially with respect to the classification of malicious versus benign activities.

This paper has the following three contributions:

- We identify string analysis as an under-explored but still quite useful type of analysis for Android apps.
- We present case studies that illustrate typical uses of strings in Android apps. These case studies also demonstrate the potential usefulness of string analysis.
- We report our preliminary results for URL, Java reflection, and Intent string analysis. We have analyzed 517 benign apps from Google Play and 1042 malicious apps from the MalGenome project [8]. We believe that our results provide enough evidence that string analysis can provide additional, deciding features that can improve flow disambiguation.

The remainder of the paper is organized as follows. In Section II we present motivating examples for URLs, Android Intents, and Java reflections. It identifies how strings are used in such operations and the value added of string analysis as well as the inherent difficulties of the approach. We present the implementation and related algorithms in Section III. Preliminary results, validation, and threats against validity are presented in Section IV. Testing of the algorithm was performed on a corpus of both malicious and benign apps. Analysis of aggregate results identifies drastic differences between the employment of strings in operations for these two types of apps. A deep dive for a selected set of results is provided. The studies reveal pertinent details about how malicious apps are engineered and how they can be identified. Related work and future directions are presented in Section V and Section VI, respectively.

II. Motivation

In Android apps strings are used pervasively to uniquely identify Android-specific constructs, to specify files and network addresses that an app uses, as well as to enumerate classes and methods invoked dynamically through the Java reflection APIs. Understanding how strings are constructed in the code base as well as the concrete values the strings contain is vital to the understanding of the apps themselves. Consider the following examples taken from real world apps, both malicious and benign.

A. URLs

Many developers make their data publicly available on Internet servers and provide an interface for extracting slices or sections of that data. URLs are heavily used in Android apps to communicate between clients and servers. Some apps gather users' phone data and send it to remote servers to request service content. Malware also takes advantage of this pattern to perform harmful actions. In order to protect users' privacy, many analyses are currently tracking what data is sent off the phone via the Internet. This, however, is not sufficient because it is not trivial to distinguish malicious behavior from benign behavior. To differentiate malicious and benign behaviors, we must not only track *what* data is, but also *where* data is sent and *how*. To do this, an analysis engine must be able to track network URLs. The following example gives us a brief idea on how a URL object is constructed and used in an Android app.

```
private void execTask(){
    ...
  str2 = "http://" + Base64.encodebook(
    "2maodb3ialke8mdeme3gkos9g1icaofm", 6, 3) +
    "/mm.do?imei=" + this.imei;
    ...
  httpString = str2 + "&mobile=" + this.mobile;
    ...
  http=BaseAuthenicationHttpClient
       .getStringByURL(httpString);
    ...
}
```

Code 1. Android URL Construction Example

In the example, the app first creates a base URL string and appends additional data (*e.g.*, IMEI, Phone Number) to construct the complete URL object. Finally, the app sends the string out via a HttpClient. Even though it is common to append data to a URL string in benign apps, there are notable differences in the above example. One key difference is the obfuscation (*how*) of the IP address of the URL (*where*) via the *encodebook* method. Malicious apps show a far greater tendency than benign apps to obfuscate the IP component. In the example, the URL object is also used to exfiltrate the mobile number and IMEI (*what*), operations benign apps do not perform.

B. Android Intents

Intent is one of the most heavily used Inter-Component Communication (ICC) mechanisms in Android. Android Intent provides a nice way for benign apps to communicate, but Android malware can also take advantage of this to perform malicious behaviors. Researchers have spent a great effort on analyzing Intent for Android ICC [9, 10]. To understand how an Android app works, we must discover the relation of components connected by intents. The following code example shows how intents are used in an app.

```
public void onReceive(Context context,Intent intent){
  if(intent.getAction().equals(
      "android.intent.action.BOOT_COMPLETED")){
    intent = new Intent("android.intent.action.RUN");
    intent.setClass(context, zjService.class);
    context.startService(intent);
  }
  ...
  while (true){
    if(!intent.getAction().equals(
      "android.provider.Telephony.SMS_RECEIVED"))
          break;
    intent = intent.getExtras();
  ...
    msg[i] =
        SmsMessage.createFromPdu((byte[])intent[i]);
    this.sms_code = msg[i].getOriginatingAddress();
    this.sms_body = msg[i].getDisplayMessageBody();
    WriteRec(context, "zjsms.txt", this.sms_code +
        "#" + this.sms_body);
  ...
  }
}
...
private void uploadAllFiles(){
  ...
  upload.uploadFile("http://"
  + getKeyNode("dom","dom_v")
  +"/zj/upload/UploadFiles.aspx?askId=1&uid="
  + getKeyNode("uid","uid_v"),
  "/com.creativemobi.DragRacing/files/zjsms.txt");
   ...
}
```

Code 2. Android System Intent Example

This example contains three types of intents. Firstly, the app catches the system BOOT_COMPLETED intent and starts its service zjService. Additionally, it monitors two other system intents: SMS and OUTGOING_CALL. We only show the SMS intent in the example. As observed from the example, each intent has its own unique string which the program uses to differentiate intents. There might be a large number of intents contained in a single app. Thus, it is important to not only identify all intents, but to infer connections between different program components in order to understand the behavior of the app. This is especially important for Android because many Android malware apps take advantage of system intents to start malicious code. The above example is from one of the known Android malware. By monitoring the system intents, the app catches the incoming SMS and Phone Call and writes to local files. Later, it uploads these files to a remote server in the zjService. To understand the complete execution flow above, we must discover all the strings that define an intent first, and then build up the correct flow of the program, which is needed for future analysis (*e.g.,* data flow analysis).

C. Java Reflection

Reflection is commonly used by programs to provide runtime support and is used pervasively in Android. Since reflection is a dynamic feature, it is usually not handled by static analyses. However, precision of static analysis is critically hurt if reflection is ignored completely. In this case, it is important to find the reflection statically defined by strings. The following Jimple (intermediate representation used by Soot and our tool) code example shows how reflection is used in Double Twist, a popular music player available in Google Play. Due to space constraints, we refer to Vallée-Rai et al. [11] for more details on Soot and Jimple.

```
$r4 = <com.doubleTwist.util.l: String
    a(String)>("c2VuZEJpbGxpbmdSZXF1ZXN0\n");
<com.doubleTwist.androidPlayer.lr: String i> = $r4;
$r4 = <com.doubleTwist.util.l: String
    a(String)>("c2VuZEJpbGxpbmdSZXF1ZXN0\n");
$r5 = <java.lang.Class: Class
    forName(String)>($r4);
$r2 = <com.doubleTwist.androidPlayer.lr: String i>;
$r6 = newarray (Class)[1];
$r6[0] = class "android/os/Bundle";
$r7 = $r5.<Class: reflect.Method
    getDeclaredMethod(String,Class[])>($r2, $r6);
$r8 = <com.doubleTwist.audio.AudioVolumeService:
    Object e()>();
$r9 = newarray (Object)[1];
$r9[0] = $r3;
$r8 = $r7.<java.lang.reflect.Method: Object
    invoke(Object,Object[])>($r8, $r9);
```

Code 3. Android Reflection Example

As observed above, the app loads a reflection object using an encrypted string. Then, it fetches the wanted method from reflection by using another encrypted string and executes it by calling `invoke`. Clearly, it performs some abnormal actions as it obfuscates the class and method names loaded via reflection. This is one of the common patterns in malware we have observed during our string analysis.

III. Implementation

Our string analysis prototype is being built into BlueSeal, a data flow analysis framework we have developed [3]. It is publicly available, along with data sets, experimental results, and plotting scripts for download at http://blueseal.cse.buffalo.edu. BlueSeal itself is built on the Soot [11] bytecode optimization framework. Our string analysis runs three phases as follows.

A. Interprocedural Analysis

The first phase of our tool gathers relevant information on strings used in intents, files, URLs, and Java reflection. This is done through a backward inter-procedural flow analysis originating at relevant invoke statements of API calls, which the tool takes as input. The analysis itself is mostly standard, but leverages functionality from BlueSeal to recreate the Android app's call graph. The most important feature of our implementation is the capability to reconstruct complex strings. That is, it can reconstruct strings built by multiple string-related operations across different methods. This capability would dovetail nicely with other tools such as IccTA [10] that performs single method string analysis. Due to the structure of Android apps, not all method calls are in the default generated call graph. For more details on call graph reconstruction for Android, please refer to our previous work on BlueSeal [3].

B. Abstract Interpretation

The output of our first phase is a set of backward flows, where each flow is represented as a graph starting with a single root. This root for a backward flow graph is a string argument used in intents, files, URLs, or Java reflection. Thus, each graph is effectively a backward *slice* of a program that consists of the code used to construct a string of interest. This gives us an opportunity to analyze the *structural* properties of how strings are created and used; we perform an abstract interpretation over each graph in order to do this. As the results presented in Section IV show, in many cases this allows us to generate concrete string literals. Our abstract interpretation engine understands basic Java string construction APIs including the `StringBuilder` class.

C. String Classification

Once a graph is generated, we classify strings into multiple categories. The goal of classification is to provide multiple types for strings which eventual clustering or disambiguation algorithms could use to classify a string as benign or malicious. Currently, we classify strings into three categories as follows.

- **Plain strings:** This category includes strings constructed using `String` and `StringBuffer` classes. These are perhaps the most frequently used classes to build a string and include convenient methods such as `replace`, `append`, and `substring`.
- **Derived strings:** This category includes strings originated from sources other than string literals found in the code. For example, strings originated from `toString`, `Class.forName`, or a string read from a configuration XML file all belong to this category.
- **Intraprocedural or Interprocedural strings:** This category identifies if a string is created within a single method or across multiple methods.

In the future, we plan to test a number of different categories based on the inferred properties and code characteristics that our tool is currently able to extract. We currently count the number of class and static fields used in the creation of the string, the number of methods called in the creation of the string, especially where aggregated strings show deep nesting of method invocations to retrieve string parts, and graph patterns for constructed strings. The latter characteristic offers the possibility to match obfuscated code against known graph patterns to determine likely intent and semantics of operations performed.

IV. Results

Development and initial validation of the algorithm was performed using unit test and gold standard Android apps obtained from GitHub. Unit test APKs were designed to test the validity of the inter-procedural algorithm, especially with respect to chaining string creation graphs for parameter and return values. Precision and recall metrics were developed for

the gold standard GitHub apps with respect to total number of URL objects created and for each created object the number of method calls that contributed to the string's creation.

The algorithm was next tested on a small subset of the MalGenome apps. Validation of results for MalGenome required a different test methodology, as source code was not available. All URL objects were identified by inspection of Soot Unit objects. Manual tracing of string creation was performed to identify and resolve issues with the string analysis algorithm. The initial version of output formats used in the following aggregate analysis section were developed and perfected to capture necessary information. Manual inspection of MalGenome apps solidified the string analysis algorithm implementation and this inspection process continued as analysis results identified the need to capture additional information for customized output formats.

A. Benchmarks

Two sets of Android apps were used for benchmarking; the MalGenome corpus of malicious apps consisting of 1042 apps and the Google Play corpus consisting of 517 apps. Both sets complement one another as they contain a broad spectrum of apps from those relatively small in size (hundreds of KBs) to larger apps that are in the tens of MBs. The string analysis algorithm was fitted with a threaded driver class that allowed for multiple, concurrent analysis executions. The MalGenome corpus was processed in six hours and Google Play in ten hours using a thread pool of size five. Interesting to note is that across all equally-sized apps, Google Play apps take longer to process than malicious apps.

Multiple output formats were generated to enable the aggregate analysis of results. These included a tabular view of all strings detected by the analysis algorithm, a graphical view of the Soot Units used in string creation, values and counts of all reflection and URL string values across all apps, and counts of all method calls across all apps. These four output types provided a great deal of information about string creation and contributed to identification of the high level observations discussed in the following subsections.

1) Creation of URLs: URL creation presents a wealth of information that may be used to categorize the behavior of MalGenome and Google Play apps. Consider Table IV-A1, which shows our tool's results for URL parameters for both MalGenome and Google Play apps. Provided are the top ten parameters identified for URLs constructed in these apps along with counts (Note that for brevity, we only present `java.net.URL` objects). Immediately it is evident that while Google Play apps provide information about timestamp, platform, and app ID, malicious apps are interested in passing information with respect to IMSI, ICCID, and telephone number. Some overlap does exist between arguments where &timestamp and &p1 trace to Java Date objects used to collect the timestamp when the URL is sent.

The tool captures URLs that are discrete strings as well as strings that are a series of appends. Analysis on the discrete strings yields a simple observation summarized in Table IV-A1. For the 1327 distinct, discrete strings identified for Google Play apps, 83 used https. For the 413 identified for MalGenome, only 3 used https. All three cases were calls to mainstream URLs—Facebook and MySpace. This simple observation demonstrates that the authors of malicious apps are unconcerned with encryption of the actual transfer of content.

TABLE I. TOP URL PARAMETERS

MalGenome		Google Play	
Parameter	**Total**	**Parameter**	**Total**
&p1=	208	&c=	115
&imsi=	109	&uid=	48
&ca=	99	&game=	33
&ac=	99	&udid=	29
&shid=	99	&app_id=	27
&err=	98	&s=	22
&tel=	96	&timestamp=	22
&iccid=	93	&sid=	21
&pid=	87	&locale=	21
&sim=	75	&platform=	20

TABLE II. USAGE OF SECURE AND INSECURE HTTP:// CONNECTIONS

	http://	https://
Google Play	1244	83
MalGenome	410	3

While unconcerned with encryption of the transfer of content, malicious app developers are extremely concerned with encryption of the IP addresses they connect to. An examination of Google Play apps revealed that there were no instances of obfuscated or encrypted URL usage. The same examination of MalGenome apps identified hundreds of instances of obfuscated or encrypted URL usage. Table IV-A1 demonstrates this, where the left column are methods that accept obfuscated text (typically the IP portion of the address) and are used in URL string creation. Zhou *et al.* [12] discuss some of these methods in greater detail.

There are numerous observations related to this table. The Example column indicates that obfuscation results tend to fall into two categories, those starting with the sequence 'Hoip' and those that do not. This categorization corresponds to the method signatures in the Method column where all methods except `encode` and `encodebook` use 'Hoip' strings.

2) Reflection: We have run our string analysis on both known MalGenome malware and Google Play apps to gather statistics on how malware and legitimate apps leverage `Reflection`. In Table IV-A2, we show the gathered statistics. Here, we only show the most common reflection strings based on the occurrence count. We observed that the most commonly used reflection strings are related to the Ads classes. As we manually examine the apps, it turns out some of the apps leverage reflection to request Ads from different libraries. Another family of interesting reflection strings is found in what appears to be an official Apache HttpClient library used by some apps. In this library, reflection is heavily used in many different places. However, after comparing the code to the official Apache library code, we have discovered that these two are completely different as the official library never uses reflection. The last interesting class of apps that use reflection involves a known malicious family called `Geinimi`. By examining the source code manually, we have discovered that the `Geinimi` code uses reflection to start a malicious service, which steals sensitive data, including `SMS`, `Device ID` and `Phone Number`, and sends it to a remote server.

TABLE III. ENCRYPTED URL USAGE

Method	Total Calls	Spread	Example
⟨com.keji.danti.util.ap: String a(String)⟩	489	163	HoiprJbh9C519IF5HxiL9I0h8cMNuezDrebh7
⟨com.sec.android.providers.drm.Xmlns: String d(String)⟩	356	178	HoiprJbh9CVN9wnQ0w7O84FePwnYPJShH
⟨com.android.main.Base64: String encodebook(String,int,int)⟩	264	22	kl4ofgsmgeje5gko99s1fc2ofm
⟨com.sec.android.providers.drm.Onion: String a(String)⟩	178	178	HoiprJbh9CVp9I0h8Cg1zKVO7CAO7CfaPJSQ
⟨com.keji.util.pd: String a(String)⟩	126	42	HoiprJbh9NDs9I0h8Cg1zKVO7CAO7CfaPJSQ
⟨com.keji.util.pf: String d(String)⟩	120	40	HoiprJbh9C519IF5HxiL9I0h8cMNuezDrebh7
⟨com.android.battery.a.pf: String d(String)⟩	43	43	HoiprJbh9C519IF5HxiL9I0h8cMNuezDrebh7
⟨com.android.main.Base64: String encode(String,int)⟩	16	2	alfo3gsa3nfdsrfo3isd21d8a8fccosm
⟨com.android.Base64: String encode(String,int)⟩	8	2	alfo3gsa3nfdsrfo3isd21d8a8fccosm
⟨ac: String d(String)⟩	4	2	HoiprJbh9C519IF5HxiL9I0h8cMNuezDrebh7
⟨com.android.sf.adomb.Transitional: String d(String)⟩	2	1	HoiprJbh9CVp9I0h8Cg1zKVO7CAO7CfaPJSQ

TABLE IV. REFLECTION USAGE IN ANDROID

Reflection Class String	Count
com.admogo.*(Ad)	2323
com.waps.*(Ad)	434
org.apache.commons.httpclient.*	369
com.geinimi.custom.GoogleKeyboard	342
com.google.ads.AdView	163

3) Intent: Analysis was performed on intents for both MalGenome and Google Play with 4000 intent calls in the former and 10000 calls in the latter. Results were analyzed by grouping the string argument values provided to intents and sorting by frequency. Table IV-A3 shows the result of this analysis.

Several interesting conclusions can be drawn from the table. The Google Play apps display an expected pattern of intent usage. The VIEW intent is indicated by Google documentation to be the most used intent with the documentation also indicating that SEND and MAIN are popular. Additionally, there are numerous intents related to billing, again expected for commercial apps.

The MalGenome apps show a very different usage pattern. They also make use of the VIEW intent as expected. Second most popular is WEB_SEARCH, used to make `URL` connections. This intent ranks 39th in Google Play with 29 usages. Next most popular is APP_DEVELOPMENT_SETTINGS, a non-mainstream intent with few references on the Internet. No Google Play apps use this intent. Next is the SEND intent, which communicates information between apps, also popular in the apps. The 5th and 6th ranked intents for MalGenome are encrypted strings. As discussed earlier in this paper, use of these strings is clearly for malicious activities and is omnipresent in intents.

An analysis of Google Play apps was performed to identify if those apps used encrypted strings. One app was identified, `com.easy.batter.saver.apk`, calling the encrypted string through `com.google.utils.NetworkUtil`. Though the namespace would indicate a trustworthy pedigree for the method, it turns out that it is not part of any standard Google API with few references to its existence.

V. RELATED WORK

To improve mobile app security and privacy, various systems have been proposed. For example, TaintDroid [13] and VetDroid [6] apply dynamic taint analysis to monitor apps and detect runtime privacy leakage in Android apps. ScanDroid [14] aims to automatically extract data flow policy from the manifest of an app, and then check whether data flows in the apps are consistent with the extracted specification. Alazab *et al.* [15] provide a dynamic analysis technique that runs apps in a sandbox and detects malicious apps. MockDroid [16] is a tool that protects users' privacy by supplying mock data instead of sensitive data. Aurasium [17] provides user-level sandboxing and policy enforcement to dynamically monitor an app for security and privacy violations. Notably, Aurasium does not require modifications to the underlying OS. CrowDroid [18] is an offline analysis over traces that can be leveraged to identify malicious apps through examining their behavior via crowdsourcing. Moonsamy *et al.* [19] provided a thorough investigation and classification of 123 apps using static and dynamic techniques over the apps' Java source code. IccTA [10] and Epicc [9] are frameworks that perform ICC analysis to detect privacy leaks that may be careless or malicious in nature. Our own work, BlueSeal [3], proposes a new permission mechanism that leverages static data flow analysis to discover sensitive data usage and improves Android app security. PiOS [20], a static analysis tool for iOS, leverages reachability analysis on control-flow graphs to detect leaks. ComDroid [21] and Woodpecker [22] expose the confused deputy problem [23] on Android.

VI. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper we presented a tool for disambiguating strings within Android apps. Our tool performs a string analysis over an Android app, creating a graph-based representation of the code blocks used in generating strings. An abstract interpretation of the graph yields a set of potential constructed strings. Our results indicate that many strings can be fully disambiguated (quantified) statically within Android apps. Reasoning about the disambiguated strings gives additional insights into the structure and purpose of an Android app.

As part of our future work, we plan to integrate our string analysis engine into BlueSeal [3], a static data flow analysis tool we have built. Based on our preliminary results, we believe that our string analysis engine will improve the precision of BlueSeal's flow analysis, specifically in identifying flows that are likely present in malicious code and, conversely, identifying when suspicious flows are in fact benign.

ACKNOWLEDGMENTS

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

TABLE V. TOP INTENT IDENTIFIERS IN ANDROID

Google Play		MalGenome	
Intent	**Total**	**Intent**	**Total**
android.intent.action.VIEW	1866	android.intent.action.VIEW	1194
android.intent.action.SEND	1038	android.intent.action.WEB_SEARCH	331
android.intent.action.MAIN	240	com.android.settings.APP_DEVELOPMENT_SETTINGS	266
android.media.action.IMAGE_CAPTURE	198	android.intent.action.SEND	198
com.android.vending.billing.MarketBillingService.BIND	151	7xBDrIM1zvBOzKlOuCRIHcBlcy09KLuFiDRd3pMccY__	179
com.android.music.musicservicecommand	150	7xBDrIM1zvBOzKlO7CMO8WB0iyiFl3MPqpzq	171
com.google.android.c2dm.intent.REGISTER	148	com.myplayer.toService	107
com.android.vending.billing.RESPONSE_CODE	141	android.intent.action.CALL	81
com.android.vending.billing.PUR_STATE_CHANGED	140	android.intent.action.MAIN	71
com.google.android.c2dm.intent.UNREGISTER	90	android.media.action.IMAGE_CAPTURE	63

REFERENCES

[1] F.-S. Labs, "Threat report h2 2013," https://www.f-secure.com/documents/996508/1030743/Threat_Report_H2_2013.pdf, 2013.

[2] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014.

[3] 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*, ser. ASE '14. New York, NY, USA: ACM, 2014.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014.

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

[6] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013.

[7] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011.

[8] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (Oakland), 2012 IEEE Symposium on*, 2012.

[9] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013.

[10] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 International Conference on Software Engineering (ICSE)*, 2015, to appear. [Online]. Available: http://www.bodden.de/pubs/lbb+15iccta.pdf

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

[12] Y. Zhou and X. Jiang, "An analysis of the anserverbot trojan," 2011.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010.

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

[15] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian, "Analysis of malicious and benign android applications," in *Proceedings of the 2012 32nd International Conference on Distributed Computing Systems Workshops*, ser. ICDCSW '12. Washington, DC, USA: IEEE Computer Society, 2012.

[16] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '11. New York, NY, USA: ACM, 2011.

[17] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: practical policy enforcement for android applications," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12, 2012.

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

[19] V. Moonsamy, M. Alazab, and L. Batten, "Towards an understanding of the impact of advertising on data leaks," *Int. J. Secur. Netw.*, vol. 7, no. 3, Mar. 2012.

[20] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *NDSS*. The Internet Society, 2011.

[21] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011.

[22] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*. The Internet Society, 2012.

[23] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, Oct. 1988.