Parameter Values of Android APIs: A Preliminary Study on 100,000 Apps

Li Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
{li.li, tegawende.bissyande, jacques.klein, yves.letraon}uni.lu

Abstract—**Parameter values are important elements for understanding how Application Programming Interfaces (APIs) are used in practice. In the context of Android, a few number of API methods are used pervasively by millions of apps, where these API methods provide app core functionality. In this paper, we present preliminary insights from *ParamHarver*, a purely static analysis approach for automatically extracting parameter values from Android apps. Investigations on 100,000 apps illustrate how an in-depth study of parameter values can be leveraged in various scenarios (e.g., to recommend relevant parameter values, or even, to some extent, to identify malicious apps).**

I. Introduction

In recent years, we have been observing the spreading of Android support for numerous devices, varying from small devices such as smart phones and watches to large devices such as fridges, TV sets, etc. This momentum of Android is accompanied by the development community who produces large numbers of apps – 1,6 million apps as of July 2015 – that users leverage in all activities of their daily life (from email and social networking to text processing and banking operations).

Unfortunately, the high market share of Android has contributed to making him a target of choice for malware writers. For example, a number of malicious apps have harmed users' finances by sending premium SMS messages with payments being collected by malware writers. In such cases, however, an analyst could easily prevent the malicious behaviour if he could check the value phone number beforehand (e.g., by extracting the value of the first parameter of SMS-sending API method *sendTextMessage*). To allow a more systematic check by any app analyst, one solution could be to build a blacklist of premium numbers by collecting, based on a large set of Android apps, all constant values that are passed to the *sendTextMessage* API method.

A recent study with the Harvester [11] approach has demonstrated that sensitive runtime data from Android apps can be used to identify malware. With the Bouncer, Google Play's maintainers are already leveraging runtime data to filter out those apps that violate the store's policy (e.g., sending SMS to their blacklisted phone numbers). Such sensitive data often involve parameter values for some sensitive Android API methods.

We argue that, besides malicious app identification, a harvest of parameter values can be leveraged to support other analyses. Consider for example an API method m, and the set s of parameter values that m uses. If s is collected from a large enough dataset of apps, one should be able, based on s, to puzzle out what are the standard uses of m in practice. Thus, building on the harvested values, one can for instance suggest appropriate parameter values to developers who would want to use m in the future. This recommendation is often essential in cases where proper documentation is lacking. As an example, let us consider API method *getLastKnownLocation(provider)* which takes a string parameter named ("provider"). In the official documentation[1], the explanation for the parameter value simply states: *"provider: the name of the provider"*, which may be insufficient for guiding developers.

In this paper, we aim to investigate to what extent parameter values can be harvested and leveraged in Android analyses. To fulfil this endeavour, we first need to extract all possible parameter values of a given set of API methods. Unfortunately, state-of-the-art works such as Google Bouncer and Harvester are not publicly available. Furthermore, both the Bouncer and Harvester are dynamic and thus require runtime execution of apps. Such approaches may not scale to app market sizes, and may also be bypassed by sophisticated malware (e.g., such malware often perform malicious behaviour at a specific time and can even hide their behavior when they detect a testing environment).

To mitigate the limitations of dynamic analysis, and to scale our investigations, we propose an approach called *ParamHarver* for harvesting API parameter values purely statically. *ParamHarver* performs backward string analysis to extract the possible parameter values of a given API method. Overall, we make the following contributions:

- a static analysis tool called *ParamHarver* that collects preset (i.e., hard-coded constant) parameter values for a given set of Android API methods.
- an evaluation of *ParamHarver*'s usability for a set of 100,000 Android apps.
- a preliminary study on the parameter values reported by the experiments performed on the dataset of 100,000 Android apps.

II. Extraction of Parameter Values

To motivate our work on *ParamHarver*, we discuss a real-world example in Section II-A. Subsequently, in Section II-B, we discuss the details for implementing *ParamHarver*, which

[1] http://developer.android.com/reference/android/location/LocationManager.html#getLastKnownLocation(java.lang.String)

```
 1 class com.shuqi.controller.BookMark$10
       implements OnClickListener {
 2   public void onClick(View view) {
 3     String[] nums = {
 4       "13720072005", "13720062005",
 5       "13701370135", "13720000800"
 6     };
 7     long tm = System.currentTimeMillis();
 8   Random r = new Random(tm);
 9   int seq = r.nextInt(4);
10   String number = nums[seq];
11   Util.sendSMS("SQ", number);
12   }
13 }
14 class com.shuqi.common.Util {
15   public static boolean sendSMS(String s0,
       String s1) {
16     SmsManager sm = SmsManager.getDefault();
17   sm.sendTextMessage(s1, null, s0, null, null);
18   }
19 }
```

Listing 1: A code snippet extracted from a real-world Android app whose package name is *com.shuqi.controller*. Some irrelevant code are excluded for simplicity reasons.

in summary performs inter-procedural string analysis to collect parameter values of Android API methods.

A. Motivating Example

Listing 1 shows a code snippet[2] taken from a real-world Android app called *com.shuqi.controller*. This snippet sends a text message through the Android API *sendTextMessage*, which include two relevant string arguments: *s1* and *s0* (line 17). In order to extract their values, backward string analysis is needed. Furthermore, both *s1* and *s0* are parameters of *sendSMS* (line 15), and are actually defined in method *onClick* (line 2). Thus, the backward string analysis must also be inter-procedural.

As shown in lines 3-6 within *onClick()*, there are actually four (premium) phone numbers defined in the array *nums*. In other words, these four numbers are possible parameter values of *sendTextMessage()*. If these numbers are successfully harvested, we could use them later on to infer whether the investigated app is likely malicious or not by simply comparing them with a set of known premium numbers (or vulnerable numbers).

B. Implementation

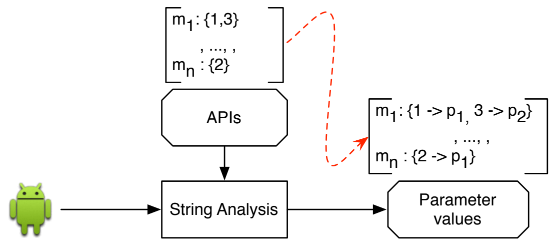


Fig. 1: The main working process of *ParamHarver*.

[2]Note that we have rewritten this snippet to Java source code in order to ease the understanding.

Fig. 1 illustrates the main working process of *ParamHarver*, which takes as inputs an Android app and a set of Android API methods along with their meta-data (configured by users of *ParamHarver*) which indicate precisely the position of interested string parameters that *ParamHarver* tracks. As an example, the meta-data for method *sendTextMessage()* in Listing 1 is $sendTextMessage : 1, 3$, requesting *ParamHarver* to harvest the value of the first and third parameters of *sendTextMessage()*. *ParamHarver* performs backward string analysis for every concerned parameters and outputs their values. The backward tracking will keep iterating until 1) the parameter value is localized or 2) the root of the control-flow graph (CFG) is reached, as it is the case for dynamically generated values. If the first constraint is met, *ParamHarver* simply outputs the identified values. Otherwise, the second constraint is met, indicating that the parameter value is not correctly extracted. In this case, *ParamHarver* returns a regular expression *(.*)*, showing that the parameter value could be potentially everything. Existing works such as the one presented by Octeau et al. [9] can be leveraged to mitigate this.

ParamHarver is implemented on top of COAL, a constant propagation language and solver [10]. Because Android apps do not have a single entry-point (e.g., *main()* in Java apps), *ParamHarver* artificially builds a dummy main method that takes into account both lifecycle and callback methods, following our previous experiences [2], [4].

III. Empirical Investigation

In this section, we evaluate *ParamHarver* empirically on a set of 100,000 real-world Android apps, in an attempt to address the following research questions:

- **RQ1:** What is the distribution of parameter values of Android API methods.
- **RQ2:** Is it feasible to recommend possible parameter values for a given Android API method?
- **RQ3:** Is it feasible to flag malicious apps based on the harvested parameter values?
- **RQ4:** Is it feasible to perform trend analysis on the harvested parameter values?

A. Experimental Setup

As introduced in the previous section, *ParamHarver* takes as inputs two artifacts: Android Apps and API methods.

Android Apps. We randomly select 100,000 Android malicious apps from a data set of over 2 million apps previously collected from Google Play and other third-party markets (e.g., appchina, anzhi, etc.) [1], [3]. In this study, we consider an Android app to be malicious if at least five anti-virus products hosted on VirusTotal[3] have flagged it as such.

Android API methods. In this experiment, for simplicity reasons (i.e., to reduce execution time, as well as manual evaluation efforts), we select the top 10 most used Android API methods for our preliminary investigation. In order to

[3]https://www.virustotal.com: A free online virus, malware and url scanner that hosts over 50 anti-virus products including AVG, McAfee, et al.

collect these top 10 API methods, we first performed a quick study based on the sampled set of 100,000 apps. The study results are shown in Table I. Note that in this study we only consider such API methods that have at least one string parameter.

TABLE I: The top 10 used Android API methods (with taking at least one string parameter).

Seq	Method Name	Class Name	Count
1	getLastKnownLocation	LocationManager	40,609
2	requestLocationUpdates	LocationManager	18,070
3	isProviderEnabled	LocationManager	8,105
4	putString	Settings$System	3,746
5	sendTextMessage	SmsManager	3,414
6	getAccountsByType	AccountManager	2,089
7	putInt	Settings$System	1,666
8	restartPackage	ActivityManager	1,272
9	setVideoPath	VideoView	956
10	getProvider	LocationManager	922

B. Overall results

Many apps in the sampled dataset of 100,000 apps do not use the top 10 API methods. Thus, we immediately filter such apps from the study.Besides, *ParamHarver* fails on some apps due to time/memory constraints.As a result, 22,681 apps are considered for further analysis.

Fig. 2 plots the distribution of the number of top-10 API methods used by the examined Android apps. The median and mean value are 2 and 3.56 respectively. We further perform a correlation study (through Spearman's rho) between the number of API methods and the size of code in apps (i.e., the size of DEX file). The spearman correlation results ($rho = -0.0046, p - value > 0.1$) indicates that these two variables are not significantly correlated.

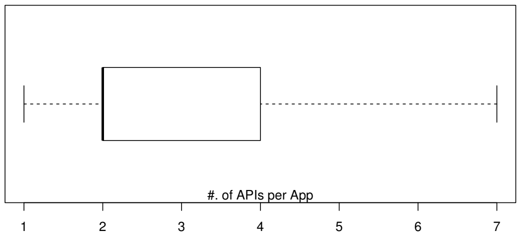


Fig. 2: The number of API methods per app without considering outliers.

C. Duplication of Parameter Values

Table II illustrates the overall results of our study. The second column indicates the position of the considered parameters for a given API, e.g., 1 means the first parameter. The third and fourth column shows the number of values harvested by *ParamHarver*, where (T) means the total number of successfully harvested values (excluding *(.*)*, *NULL-CONSTANT* and empty results) while (D) means the number of distinct values.

The huge difference between the third and fourth column suggests that most parameter values are used in duplicate. This is insightful, as most parameter values are independently used

TABLE II: The investigation results.

Method Name	Parameter Seq	Count (T)	Count (D)
getLastKnownLocation	1	32,832	3
requestLocationUpdates	1	11,691	5
isProviderEnabled	1	7,985	6
putString	2	1,330	33
	3	185	11
sendTextMessage	1	344	65
	3	6,909	150
getAccountsByType	1	2,034	8
putInt	1	1,666	24
restartPackage	1	187	48
setVideoPath	1	22	8
getProvider	1	127	5

by many different apps, we would expect that this situation happens only in API methods whose parameter values are defined in advance, e.g., *getLastKnownLocation()*. However, we would not think it likely that this also happens in other API methods whose parameter values are strongly correlated to developers' requirements. Let us take *sendTextMessage()* as an example, it has two parameters (the first one indicates the target phone number and the third one indicates the message context) that are totally correlated to developers' requirements. In our observation, phone number *106909990999*, as the first parameter, has been used 43 times. message *cxye*, as the third parameter, has been used 127 times.

We further investigate the reason why *106909990999* is used as a target phone number by 43 apps to send SMS. Interestingly, among the 43 apps, we collect seven distinct package names, meaning most of them are actually variants of same apps. Even there are seven distinct package names, all of them are signed by one of two distinct certificates. Additionally, those 43 apps shared the same prefix (*com.feedov*) for their package names. Based on these evidences, we can observe that **duplicated parameter values could be leveraged to find app variants.** The investigation results on message *cxye* has also confirmed this finding.

D. Recommendation of Parameter Values

Table III comparatively summarizes the parameter values that are harvested by *ParamHarver* on the four methods of class *LocationManager*. The three most used values (in gray) are powered by the basic Android SDK while the remaining values are powered by third-party services. The most used values, without any doubt, are "gps" and "network". "go2map" is the least used value, which is used only one time.

Being the fact that parameter values in Android API methods are usually used in duplicate, we believe our approach could be leveraged to recommend possible parameter values for API methods. Recall that in the introduction we have shown the insufficient documentation of Android API methods, where it is difficult to know what should be the parameter value of the API method *getLastKnownLocation()*. Now, with our approach, we could give a hint to developers that the first three values ("gps", "network" and "passive") are relevant to their needs, although the other four values are also possible. Furthermore, "gps" or "network" are even more applicable than "passive", as indicated by the huge difference of used times among them.

TABLE III: Comparative results of LocationManager-related API methods.

Method Name	gps	network	passive	lbs	go2map	MapABCNetwork	AutonavicellLocationProvider	Total
getLastKnownLocation	15,263	17,550	19	0	0	0	0	32,832
requestLocationUpdates	3,714	7,916	9	51	1	0	0	11,691
isProviderEnabled	6,469	1,494	1	17	0	1	3	7,985
getProvider	53	56	0	6	0	3	9	127
Total	25,499	27,016	29	74	1	4	12	52,635

TABLE IV: The harvested phone numbers and their status (C = Count, S = Status, ✓ = premium, ✗ = non-premium, ?= unknown).

Number	C	S	Number	C	S
10086	129	✗	1066185829	2	✓
106909990999	43	✓	1066953930	2	✓
10659811002	14	✓	3354	1	?
10001	13	✗	3353	1	?
7132	12	?	18703750375	1	✗
12114	10	✗	15919831203	1	✗
10010	7	✗	15860282110	1	✗
1065515810002	6	✓	+19494368398	1	?
106588003013891	6	✓	0	1	?
1065889915	6	✓	04800	1	?
13811558614	6	✗	10086999	1	✗
15919479044	6	✗	1065-5021-80133	1	✓
1065843601	5	✓	1065-71090-88877	1	✓
1065880004	3	✓	1065-9020-5111-191	1	✓
10660596	3	✓	1065505796084	1	✓
1066156686	3	✓	10655133	1	✓
13701370135	3	✗	10655576	1	✓
13720000800	3	✗	10657525748900014	1	✓
13720062005	3	✗	106580808	1	✓
13720072005	3	✗	10658422	1	✓
1065502182177	2	✓	10659057110094	1	✓
10657109050762	2	✓	1066017801	1	✓
10658368	2	✓	10669079	1	✓
10658424	2	✓	123987651017	1	?
1065902163958	2	✓	13564332483	1	✗
106601412004	2	✓	13640534425	1	✗
1066057101	2	✓	15239463832	1	✗
1066057103	2	✓	15859268161	1	✗
15645027999	2	✗	400	1	✗
13823308135	2	✗	4860008	1	?
13646870394	2	✗	7838613121	1	?
106901952345	2	✓	9494367679	1	?
1069003801012038	2	✓			

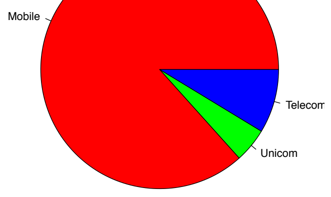
E. Parameter Values for Malware Identification

As indicated by [11], some sensitive runtime data (or parameter values of API methods) in Android apps can actually be used for identifying malicious apps. Indeed, as shown in Section III-C, many sensitive parameter values are actually used in many samples, making them good candidates for blacklisting.

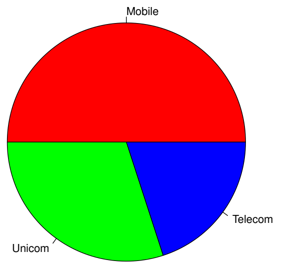
Let us consider method *sendTextMessage()* as an example. Malicious apps could use it to silently send SMS to premium numbers, causing a high financial harm to users. In our investigation, we have harvested 65 phone numbers, which are listed in Table IV. To verify whether our findings are able to identify other malicious apps, we take the list of numbers in Table IV as a blacklist to evaluate the MalGenome project [13]. With this simple setting, we are able to flag 158 (out of 1,260) apps as malicious. Table V illustrates more details on our findings, where we classify our findings through their family labels. Interestingly, only based on a blacklist of phone numbers, we are able to flag malicious apps from 15 families. Additionally, all apps of 8 families (out of 15) are fully identified.

TABLE V: Malware identification results.

Family	Count	Numbers
DroidDream	2/16	10001
YZHC	**22/22**	10086
zHash	**11/11**	10655133,10001,10086,10010
Geinimi	5/69	3353,10001,13564332483
FakePlayer	**6/6**	7132,3353,3354
Asroot	1/8	10001
BaseBridge	46/122	10001,10086
BeanBot	**8/8**	10086
Pjapps	21/58	10086999,10086,10010
HippoSMS	**4/4**	1066156686
Bgserv	**9/9**	10086,10010
DroidDreamLight	1/46	10086,10010
RogueSPPush	**9/9**	10086
Zsone	**12/12**	10655133,1066953930,106601412004, 1066185829,10086,10010
NickySpy	1/2	15859268161



(a) Normal number.



(b) Premium number.

Fig. 3: Distribution of the usage of phone numbers in different service providers of China.

F. Parameters for Trend Analysis

After harvesting a large set of parameter values, we could based on them to perform trend analysis. we now show an example of trend analysis that we observed through the harvested phone numbers.

In china, mainly, there are three service providers (SP)[4]: 1) China Mobile, whose normal service numbers start from *10086*, while premium numbers start from *10657* or *10658*; 2) China Telecom, whose normal service numbers start from *10001* while premium numbers start from *10659*; and 3) China Unicom, whose normal service numbers start from *10010* while premium numbers start from *10655*[5]. Based on these information, we manually evaluated the harvested numbers (cf. Table IV) and classified them into three categories: 1) known

[4]https://en.wikipedia.org/wiki/Telecommunications_industry_in_China.
[5]http://www.miit.gov.cn/n11293472/n11293832/n11293907/n11368223/n12977312.files/n12977310.doc

premium numbers (✓); 2) known non-premium numbers (✗), note that these numbers however could also be used for malicious purpose as all of them are harvested from malicious apps; and 3) unknown numbers (?), for which their statuses need to be further investigated.

Fig. 3 presents the distribution of the use of service numbers in different SPs: the normal service is in Fig. 3a while the premium service is in Fig. 3b. Both figures have shown that there are more malicious apps that target China Mobile than the other two SPs, suggesting that China Mobile occupies more marketing shares than the other SPs. This trend is actually reflecting the real situation, where China Mobile is currently dominating the Chinese SP market shares. Finally, we note that both Telecom and Unicorn account for less than 25% for normal numbers but increase to over 50% for premium numbers. This result suggests that malware writers tend to diversify their SP targets.

G. Threats to Validity

At the moment, our approach is unaware of obfuscation, which may lead to incomplete results. Like many other static analysis approaches, our approach is unaware of native code, multi-threads and reflections. Besides, our static analysis is also unaware of the inter-component communication (ICC) mechanism. Other approaches could be leveraged to mitigate these kinds of threats (e.g., IccTA [4] for the ICC mechanism). So far, we perform string analysis for parameters, in our future work, we plan to extend this work to also support the analysis of object parameters.

IV. Related Work

There are several approaches closing to our work. One of the most sophisticated ones is Harvester [11], which is dedicated to extract runtime values from any position in the Android code. Although it is not the focus, Harvester could be also leveraged to extract the parameter values from all the Android API methods in Android apps. However, the implementation between Harvester and our tool *ParamHarver* is quite different. As Harvester needs to execute the apps while *ParamHarver* is a pure static approach which does not need to really launch the apps.

Another related work is Precise [12], which recommends parameters of API methods automatically so as to ease the use of API methods, as the API documentations are usually incomplete. Precise mines existing code base (through certain scenario) to build a parameter usage database and then based on it to provide on-demand queries for parameter candidates through concrete usage patterns. Our work can be taken as a simplified version of Precise, based on the harvested practical parameters, we are also able to recommend the appropriate parameter candidates.

There are a batch of works that investigate the Android API methods [6]–[8] or libraries [5]. Although those works are not focusing on the parameters of API methods, we believe that our findings (the harvested parameter values) can be useful for them. For example, a suddenly increase of parameter values (starting from a specific API version) would suggest that the API or library is probably updated (e.g., to provide more functions).

V. Conclusion and Future Works

In this paper, we have first presented a tool called *ParamHarver* to automatically harvest parameter values of Android API methods. Then, we launched *ParamHarver* on a set of 100,000 apps with 10 top used Android API methods. In the next step, we perform a preliminary investigation on what we can gain from the harvested parameter values, from which we show that 1) parameter values could be leveraged to find app variants; 2) parameter values harvested from a large set of apps could be used to recommend appropriate values for practical use of API methods; 3) parameter values of some sensitive API methods can be used to identify malicious apps; and 4) parameter values, to some extend, can be leveraged to perform trend analysis.

Acknowledgments

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289.

References

[1] Kevin Allix, Quentin Jérome, Tegawende F Bissyandé, Jacques Klein, Radu State, and Yves Le Traon. A forensic analysis of android malware–how is malware written and how it could be detected? In *COMPSAC'14*.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[3] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *QRS*, 2015.

[4] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[5] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An Investigation into the Use of Common Libraries in Android Apps. In *SANER*, 2016.

[6] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *FSE'13*.

[7] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *ICPC*, 2014.

[8] Tyler McDonnell, Bonnie Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *ICSM*, 2013.

[9] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *POPL*, 2016.

[10] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *ICSE*, 2015.

[11] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[12] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In *ICSE*, 2012.

[13] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *S&P*, 2012.