HAL
open science

Information Flow Policies vs Malware

Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong, Thomas Saliou

To cite this version:

Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong, Thomas Saliou. Information Flow Policies vs Malware. [Technical Report] 2013. hal-00862468

HAL Id: hal-00862468

https://inria.hal.science/hal-00862468v1

Submitted on 16 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Information Flow Policies vs Malware

Radoniaina Andriatsimandefitra
Thomas Saliou
Valérie Viet Triem Tong
EPC CIDRE
SUPELEC/INRIA/CNRS/University of Rennes 1
firstname.lastname@supelec.fr

Abstract

Application markets offer more than 700'000 applications: music, movies, games or small tools. It appears more and more difficult to propose an automatic and systematic method to analyse all of these applications. Google Bouncer [1] tries to keep malicious applications out of Google Play by analysing uploaded applications to find known malware and malicious behaviours. However, Google Bouncer suffers from the same drawbacks of usual scan methods: it is inefficient to detect unknown malicious behaviour and it may be costly. In this paper we propose another method to efficiently detect malicious actions of applications. Our proposal consists in a new scheme of submitting applications to market place and installing applications on the device. More precisely, applications are uploaded with a companion information flow policy. A companion policy exactly describes where data used by the application can flow. The policies are studied for acceptance by reviewers. Accepted policies are certified by the market and are made publicly available. When a user acquires an application, he has to retrieve the certified version of its companion flow policy. The companion policy of the application is composed with the current flow policy enforced in the system. The application is then monitored and each time the monitor detects an information flow not allowed in the composed flow policy it raises an alert or blocks the information flow. This way, only applications respecting an official policy accepted by the market can efficiently run.

1 Introduction

During the last years, the use of information flow monitoring and control on mobile environment has been booming. In [2] Enck et al presented TaintDroid and exposed the threat, proving by example, that more than a third of the most popular applications available on *Google Play* are responsible for sensitive information leakage. More recently Gu et al presented in [3] D2taint a system able to track system data flows for Java code at the variable level, similar to

what TaintDroid does. D2taint goes further than TaintDroid since its tagging strategy permits numerous information sources in multiple classes at runtime. TaintDroid or D2taint are information flow monitors that observe exchanges of information in the Dalvik virtual machine and thus are aware of information exchanges between Java applications. Inherently, these kinds of monitors are blind when leakages occur out of Java applications. In [4], Yin et al. use taint tracking at hardware level in an emulated environment running Microsoft's operating system Windows to detect and analyse malware's behaviour. In continuous works [5], Yan et al. perform the same kind of studies for malware infecting Android platforms. However, in [4, 5], the environment in which the operating system runs is emulated. In [6], we have designed and experimented a concrete and detailed information flow policy dedicated to mobile environment. For these experiments we have used Blare[1], an information flow monitor able to intercept system calls in order to observe information exchanges in the whole operating system. In this work, we explore the use of information flow policies in order to achieve detection of malicious versions of applications.

More precisely, our proposal is structured around the three following points. First, we expect that developers of applications construct relevant flow policies for their applications and we propose here a semi-automatic method to help them. Second, we propose that market owners publish and sign all information flow policies they have authorized on official markets. Third, when a user acquires an application, he enforces the companion flow policy using an information flow monitor. If a user inadvertently acquires a version of an application infected by a malware (from an *unofficial* market for instance), we suggest him to retrieve and to enforce the official version of the flow policy. This way if the unofficial application is infected by a malware, its malicious actions are detected by the information flow monitor. We claim that this protocol permits to detect only information flows induced by unofficial applications which reveal helpful to detect malicious information flows. The main problem lays in the construction of a precise and relevant information flow policy. This paper explores a possible response to these problems and verifies its relevance. We use the *Blare Security Policy Language* (BSPL) a language that permits to specify precise flow policies [7]. BSPL permits to define precisely expected behaviour regarding sensitive pieces of information, and to compose several information flow policies.

In the following, we explore this proposed scenario. First, we briefly present the underlying information flow model (section 2) and the corresponding monitor that is in charge of enforcing an information flow policy of the previous detailed model. In this work, policies are expressed in the BSPL language. We present the main features of this language and detail the implementation of a BSPL policy manager for android devices (section 3). In section 4, we explain how taint graphs can help to automate the process of a flow-policy creation. Finally, in section 5, we evaluate the effectiveness of the policies. More precisely, we test if the execution of an application causes alerts that violate its com-

[1] `http://blare-ids.org/`

panion flow policy. We also test if the monitor succeeds in detecting malicious information flows induced by infected version of the same application.

To sum up, the contribution of this article is threefold: it presents a BSPL policy manager for android devices that permits to compose and apply information flow policies, it gives a semi-automatic method to build such policies, lastly it presents some experiments that evaluate efficiency of policies to cope with malware.

2 The underlying information flow model

We conform here to the model of information flow policy previously detailed in [8, 9, 6]. In this model, an information flow policy is defined in three main steps. The first step consists in defining the sensitive pieces of information that have to be protected in the environment. A unique numerical identifier is associated to each of these pieces of information. The second step consists in defining in which information containers and with which other pieces of information these pieces of information are allowed to flow. The policy is a set of pairs on the form $(c, \{\{i_{11} \dots i_{1n}\} \dots \{i_{m1} \dots i_{mn}\}\})$ where c is a container of information (which can be either a file, a socket, a process) and $\{\{i_{11}, \dots i_{1n}\}, \dots \{i_{m1}, \dots, i_{mn}\}\}$ is a set of sets of information identifiers. A pair of this form expresses that any data computed from any piece of information that is a subset of a set appearing in $\{\{i_{11}, \dots, i_{1n}\}, \dots \{i_{m1}, \dots, i_{mn}\}\}$ is allowed to flow into the container c. Such information flow policies can be checked at system level using an information flow monitor as explained below. In this work we use the Blare monitor [6, 9], that uses two tags, namely `itag` and `ptag`, attached to each information container. For a given container `c`, `c.itag` is a set of information identifiers that expresses from which tainted content the current content of c has been computed. `c.ptag` expresses the element of the policy that concerns `c`. In other words, if $(c, \{\{i_{11}, \dots, i_{1n}\}, \dots \{i_{m1}, \dots, i_{mn}\}\}$ belongs to the information flow policy then `c.ptag` is equal to $\{\{i_{11}, \dots, i_{1n}\}, \dots \{i_{m1}, \dots, i_{mn}\}\}$ meaning that `c` is authorized to contain any non marked content or any mix of tainted data mentionned at least in a subset of $\{\{i_{11}, \dots, i_{1n}\}, \dots, \{i_{m1}, \dots, i_{mn}\}\}$. For instance, consider that `c` is the process `surfaceflinger` and that `surfaceflinger.ptag` is equal to $\{\{1, 2\}\{3\}\}$ where i is an information identifier associated with the android package file (APK) of an application i. If applications 1 and 3 request at the same time to draw something on the screen, then they will use the service `surfaceflinger` whose itag will thus contain at least $\{1, 3\}$. This itag value is not included in any subset of `surfaceflinger.ptag` which will lead the monitor to raise an alert.

Blare monitors information flow occurring between information containers at system level such as files, process, sockets. Blare implementation is based on the *LSM* framework [10] that offers a way to completely capture information flows induced by system calls. Blare has also a set of hooks specific to Android that makes Blare precisely aware of information flows occurring inside the *binder* an Android specific Inter Process Communication mechanism.

In this work, we use the Blare Security Policy Language (BSPL for short) [7] to specify a policy. For that purpose we have developed a BSPL policy manager dedicated for android devices. Using BSPL, an application developer can define its own pieces of sensitive data and where they are allowed to flow. The developer is also requested to detail which mix of data he accepts in its own containers. Information flow policies defined using BSPL can be verified, composed and applied in a Android device using the BSPL policy manager. To apply a policy, the BSPL policy manager computes and sets `ptag` on concerned files and applications. Once applied, the Blare monitor is in charge of detecting any policy violation. In the following section, we present a BSPL policy manager dedicated to android devices.

3 BSPL policy manager

BSPL is a xml based language that have been proposed in [7] whose main objective is to offer a clear, simple and univocal way to specify an information flow policy. A policy written in BSPL is composed of two main elements: the first element (called `data_policy`) identifies sensitive data that have to be protected, the second element (called `container_policy`) describes the information containers and which mix of sensitive data they are allowed to contain. Each application may define its own policy that will be composed with current policy which has itself been obtained from composition of the system policy and with policies of other applications previously installed on the device. BSPL defines a consistency property that expresses that if a mix of data is allowed to flow into a container then this container is also authorized to contain this mix of data. In [7], authors have shown that this property holds true for the composition of consistent policies.

We have developed a BSPL policy manager dedicated to Android operating system. The policy manager is an android application that involves a sqlite database used to store the current version of the policy enforced on the device (denoted by $\mathbb{P}$). The database is initialized with a system policy that defines rules for default android containers. For instance, the system policy expresses that containers used to provide services are authorized to contain any tagged data, it could also expresses that some containers are forbidden to contain any tagged data. The policy is updated each time a new application having a companion policy is added to the system. The manager is able to verify that a given policy is consistent or not. If a policy is consistent it can be composed with $\mathbb{P}$ which leads to a new value of $\mathbb{P}$. The policy manager makes some verifications on the policies before acceptance and composition. For instance, it verifies that a container (or similarly a sensitive piece of data) is uniquely defined. Applications are thus unable to declare being the owners of containers or pieces of data that already exists on the system. In case of conflict during the policy composition, the less permissive choice is retained. Lastly, the policy manager is in charge of applying $\mathbb{P}$ on the device. The respect of the policy $\mathbb{P}$ is then ensured by Blare.

4 Defining a BSPL policy using taint graph

An information flow policy (as previously detailed) suffers from its advantage. On the one hand it offers a way to precisely describe legal and, implicitly, illegal information flows in a running operating system. On the other hand it seems to be impractical because of the difficulty to construct such policies for real applications. It can become time-consuming for application developers as it requires a precise knowledge of how information is used in the whole system. To circumvent this limitation of the policy definition, we propose to build the flow policy of an application using its taint graph. Indeed, a taint graph gives a representation of how an application disseminates its own data in the system. We play the role of a developer who wants to construct a flow policy to make it approved on an official marketplace. Once developed, the application is marked using a unused identifier and run on a smartphone monitored by Blare which logs all information flows that are engendered by syscalls and involve marked data. A log entry describes the pieces of sensitive information involved in the flow, its source and destination and is labelled with a timestamp. Source and destination are described by their type (file, process or socket), their name and their system identifier. From the log entries, we construct a taint graph that describes how a running code disseminates its own pieces of information in the operating system. The taint graph permits to construct the companion policy of the application since it exactly details which containers of information the application will contaminate. We have developed a small python script that parses the log file, compacts it into a taint graph and computes the derived policy. Even if information that can be retrieved from a taint graph is equivalent to information contained in the log file, a taint graph gives a more compact representation that is helpful to adjust the policy if needed. First, we generate the BSPL policy for a benign application. The resulting policy will be its companion flow policy. Before using this policy, we need to adapt it. Processes considered as legal containers of information from the application are only denoted by their name (e.g `com.android.email`) in the taint graph. We thus need to specify the name of the corresponding executable files to set the tags properly.

The policy only declares one piece of sensitive data originating from the application `apk` file and lists all information containers where these pieces of data are requested to flow. These containers are divided into two sets: those that belong to the application itself and those that belong to the system which are referenced as such. We have applied these guidelines on applications retrieved through Google Play Store. We construct the policy of each application separately. We install the application and mark its package (`.apk` file). In background, Blare monitors and logs the information flows occurring in the system. Once installed, we use the application during a learning period. During this learning period we launch all features offered to the user. We obtain a log having from 30000 to 60000 entries that leads to the construction of a companion policy with one piece of sensitive data (content of the `apk` file) and an average of 80 containers. The policy consistency is checked by the policy manager, which can compose them with the policy $\mathbb{P}$ previously enforced on the device.

Containers appearing in a policy are not only files or processes directly created, modified or accessed by the application itself. They are information containers that may have been contaminated by the application by a direct or indirect flow. For instance, if the application is responsible of the creation of a process p, such that p writes into a socket s then both p and s are considered as contaminated by the application and should appear in the policy. For the experiments detailed in section 5 we use a policy computed for a popular game involving birds that declares an amount of 78 containers. More precisely, it details 28 containers defined in the application folder that belongs to the application itself. There are mainly data stored in local cache files, a database and some shared preferences files. The remaining containers are system containers that are mentioned in the application policy but precisely declared within the system policy. Such policy may seem big but should not be considered so. The illegal containers of marked data owned by the application are the one not listed in the policy and they are much numerous as there are for instance more than 79000 regular files on a Nexus S device running a stock version of Android Ice Cream Sandwich 4.0.4.

5 Detection of malicious behaviours

So as to evaluate the efficiency of the proposed approach, we repeat a three-step experiment thrice. Each experiment is done with two versions of an application: a benign one and a malicious one. The goal of the experiments is to evaluate the effectiveness of a policy built as suggested in section 4. First, we want to quantify the number of alerts raised when the user executes a benign application and Blare is configured with the companion policy of this application. This first test permits to evaluate an idea of false positive. Second, we wanted to evaluate the relevance of the same policy when a malicious version of an application is installed and executed instead of the benign one. Does it raise any alerts? If it is so, do these alerts correspond to the execution of the malicious code ? Each step of the experiments is explained below.

First, we generate the BSPL policy for a benign application. The resulting policy will be the companion flow policy for the benign version and infected version. Second, we install the benign application used before, mark its `apk` file, configure Blare with its companion flow policy and use it as a normal user would do. It permits to verify that the flow policy do not cause any useless alert for benign applications. Third, we repeat the second step but we use the malicious version of the application instead of the benign one. It permits to evaluate the effectiveness of the policy to help Blare to detect malicious action on the system. All tests are performed with a Nexus S running Android Ice Cream Sandwich 4.0.4. At the beginning of each step, the device is reset to a default image of Android to which we added the Blare monitor and its user space tools. We also mark two applications (`surfaceflinger` and `servicemanager`) as trusted and thus do not engender any tainted flows. The reason of this choice is their role in the system combined with the way they interact with other pro-

cesses. Their role involves interactions with a lot of running applications at the same time whereas the real engendered flows are far less important than what a monitor like Blare could see. More precisely, when Blare observes that an application exchanges data with one of these services, it considers that it retrieves at least any tainted data that were previously sent to the service. This over-approximation is too coarse for these two applications as they only propose services that do not permit any information exchange between two clients. Thus, not considering them as trusted may engender a lot of false positive. We claim that this choice is clearly legitimate for the application `surfaceflinger` since this process is in charge of drawing all user interfaces on the screen. The second process, `servicemanager`, holds references of all running services in the device. Any application that wishes to use a service thus has to ask for its reference to `servicemanager`. Due to its function in the system, `servicemanager` interacts with a lot of running application and engender a lot of false positives unless marked as trusted. We have considered during our experiments that `servicemanager` is trusted and do not disseminate information within the system. To sum up, `surfaceflinger` and `servicemanager` behave as wells for information flows.

A bird game and its version infected by LeNa: For the first experiment, we started by evaluating the companion flow policy of a bird game with its benign version. The execution of the game while Blare verifies the compliance with the policy engendered no alerts. Next, we reinitialize the environment and proceed to the evaluation of the policy with a malicious version of the same game. The malicious version of the bird game is infected by a variant of *LeNa*[11]. *LeNa* is originally disguised as a legitimate application that asks the user root privileges. The acceptance of the demand triggers the malicious payload of the malware. When root privileges are granted, the application behaves properly in the point of view of the user but in background it installs a native binary that grants remote control of the device to the attacker. He can for instance launch a stealth installation of a new Android application. Our application is infected by a variant of LeNa. According to a report [11] the main difference between them is that the variant does not rely on the user to gain root access. In the variant, the malware embeds a root exploit [12] that tries to exploit a vulnerability of the system to gain root access. The danger is thus higher as it could even affect non-rooted devices. The analysis report of the original version of *LeNa* [13] indicates that the malware, once root access is granted, dumps native binaries in the device: `/system/etc/.dhcpcd`, `/system/xbin/ccb`, `/system/etc/.rild_id`, `/system/bin/dhcpcd` and `/system/bin/installd`. We then test if our monitor detects flows induced by these behaviours. We launched and played with the application using the same environment (same device, same taint identifier and of course same companion policy) and played with it. The application behaves as the benign version from the user point of view when executed. However, Blare detects malicious behaviours and raises alerts filtered down to 15 unique alerts. The alerts indicate that two processes (`logo` and `logcat`) read and write data from / into files (`logo`, `crashlog`, `flag`, `exec` and `.e1240987052d`) located in a subdirectory

of the application home directory. Even if the corresponding activities do not appear malicious at the first sight, these alerts correspond to the first step of a try to gain root privileges[12]. However, no alerts corresponding to binary dumps in `/system` as described the report about LeNa[11] reports were raised. The reason is that no such files were created or edited during the analysis of the infected application.

A finger scanner and its version infected by *DroidKungFu1*: We repeat the same experiment as before for a benign version and malicious version of a finger scanner application. We observe no alert when executing the benign version of the finger scanner while Blare verifies the compliance with its companion flow policy. We then reinitialize the environment and use this time a malicious version of the finger scanner while Blare is configured with the same policy. The malicious version of the finger scanner is infected by *DroidKungFu1*[14] which is a malware discovered in May 2011 by X. Jiang and his research team. It is capable of rooting Android phones. As for the infected bird game, the infected version of the application behaves as the original from the user point of view when it is executed. Nevertheless, Blare raises 11 unique alerts listed in table 1. As alerts represent illegal flows observed by Blare, we can directly build an early diagnosis about the nature of the intrusion by analysing them. The first lines of the alerts indicate that the application is responsible of the creation of two files in its home directory. They are `gjsvr` and `legacy` and are from the application package. Then next step seems to consist in copying data from these files into two new files: `/system/bin/gjsvr` and `/system/app/com.google.ssearch.apk`. These copies are preceded by a flow into `/proc/sys/kernel/hotplug`. According to technical reports about *DroidKungFu1*, this corresponds to a try to gain illegally root privileges which is needed by the malware to be able to write into `/system`. Once the two copies are done, the `system_server` process detects the new file in `/system/app`, reads it and spreads data in `/data/system/packages.xml`. All of this leads us to the hypothesis that the previously-read file is an Android application package that has been installed. Illegal flows from the new package reinforce that hypothesis as the `dexopt` application extracts data from it and writes to a `.dex` file in the cache-directory of Android dalvik virtual machine. A new process called `.google.ssearch` also accedes the new `.apk` and `.dex` files. The content of `packages.xml` can also confirm the installation as it contains a new entry corresponding to the application.

A knife game and its version infected by *BadNews*: We repeat one last time the experiment for the companion flow policy of a knife game with a benign and a malicious version of the application. For the benign version, Blare also raises no alerts. The malicious version is a sample of a malware named *BadNews*[15]. *BadNews* was discovered in April 2013. Some samples were present on *Google Play* and others on its Russian alternatives. Once the malicious code is executed, it starts by contacting to a Command and Control (C&C) server to request the action to perform. Based on the analysis of the application, it can receive at least six types of command: display news notification, automatically download then launch the installation of applications, update the address of

```
[BSPL_VIOLATION] process gjsvro:gjsvro 984 > file /proc/sys/kernel/hotplug 4827 > itag[-3]
[BSPL_VIOLATION] process gjsvro:gjsvro 984 > file /system/bin/gjsvr 16738 > itag[-3 3]
[BSPL_VIOLATION] process gjsvro:gjsvro 984 > file /system/app/com.google.ssearch.apk 8330 > itag[-3 3]
[BSPL_VIOLATION] process cat:cat 990 > file /system/app/com.google.ssearch.apk 8330 > itag[3]
[BSPL_VIOLATION] process gjsvro:gjsvro 984 > socket (127.0.0.1) 0 > itag[-3 3]
[BSPL_VIOLATION] file /system/app/com.google.ssearch.apk 8330 > process dexopt:dexopt 991 > itag[3]
[BSPL_VIOLATION] process dexopt:dexopt 991 > file /data/dalvik-cache/system@app@com.google.ssearch.apk@classes.dex 24632 > itag[3]
[BSPL_VIOLATION] file /data/dalvik-cache/system@app@com.google.ssearch.apk@classes.dex 24632 > process dexopt:dexopt 991 > itag[3]
[BSPL_VIOLATION] process droid.gallery3d:droid.gallery3d 995 > file /data/data/com.android.gallery3d/
shared_prefs/com.android .gallery3d_preferences.xml 57603 > itag[3]
[BSPL_VIOLATION] file /system/app/com.google.ssearch.apk 8330 > process .google.ssearch:.google.ssearch 1059 > itag[3]
[BSPL_VIOLATION] file /data/dalvik-cache/system@app@com.google.ssearch.apk@classes.dex 24632 >
process .google.ssearch:.google.ssearch 1059 > itag[3]
```

Table 1: Alerts raised when monitoring an application infected by DroidKungFu

	Alerts for official version	Alerts for infected version
Bird game	0	15
Finger scanner	0	11
Knife game	0	209

Table 2: Observed Alerts when monitoring application under BSPL policies

the C&C server and add icons on the device that redirects to a web page or an application to be installed. We installed the malicious version, applied the policy resulting from the composition of the companion policy of the original application with the system policy and played normally with the knife game. During our analysis, Blare raised 209 unique alerts. This high number of alerts is mainly due to the malware triggering the browser to further infect many containers of the device.

To sum up, we observed 0 alerts for the benign version of the applications and relevant alerts for their malicious version.

6 Conclusion

In this presented paper we have proposed a new direction in defense against malware that pollutes application market. Our proposal mainly relies on the definition of a precise information flow policy, namely the companion policy that have to be accepted and certified by official application markets. These companion policies are specified in the BSPL language and enforced using the Blare information flow monitor. We have implemented a policy manager that permits to compose BSPL policies and set them on a mobile device. We have also led some experiments that have shown the method efficiency to detect malicious versions of applications available on market places. In future works we plan to propose tools that will help market places to efficiently review submitted policies.

References

[1] http://googlemobile.blogspot.fr/2012/02/android-and-security.html.

[2] W. Enck, P. Gilbert, B. gon 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 *In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[3] B. Gu, X. Li, L. Gang, A. Champion, Z. Chen, F. Qin, and D. Xuan, "D2taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources," in *Proceedings of the IThe 32nd IEEE International Conference on Computer Communication*, 2013.

[4] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[5] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[6] R. Andriatsimandefitra, S. Geller, and V. Viet Triem Tong, "Designing information flow policies for android's operating system," in *Proceedings of the IEEE International Conference on Computer Communications*, 2012.

[7] S. Geller, V. Viet Triem Tong, and L. Mé, "Bspl: a language to specify and compose fine-grained information flow policies," in *Proceedings of the The Seventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, to appear in 2013.

[8] S. Geller, C. Hauser, F. Tronel, and V. Viet Triem Tong, "Information flow control for intrusion detection derived from mac policy," in *Proceedings of the IEEE International Conference on Computer Communications*, 2011.

[9] C. Hauser, F. Tronel, J. Reid, and C. Fidge, "A taint marking approach to confidentiality violation detection," in *Australasian Information Security Conference (AISC)*, 2012.

[10] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security module framework," in *OLS2002 Proceedings*, 2002.

[11] http://goo.gl/kFwC6.

[12] http://c-skills.blogspot.fr/2011/04/yummy-yummy-gingerbreak.html.

[13] http://goo.gl/qoJjI/.

[14] http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html.

[15] https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/.