information

MDPI

Article

Android Collusion: Detecting Malicious Applications Inter-Communication through *SharedPreferences* †

Rosangela Casolare [1,*,‡], Fabio Martinelli [2,‡], Francesco Mercaldo [2,3,*,‡] and Antonella Santone [1,‡]

1 Department of Biosciences and Territory, University of Molise, 86090 Pesche (IS), Italy; antonella.santone@unimol.it
2 Institute for Informatics and Telematics, National Research Council of Italy, 56121 Pisa, Italy; fabio.martinelli@iit.cnr.it
3 Department of Medicine and Health Sciences "Vincenzo Tiberio", University of Molise, 86100 Campobasso, Italy
* Correspondence: rosangela.casolare@unimol.it (R.C.); francesco.mercaldo@unimol.it (F.M.)
† This paper is an extended version of our paper published in the 34th International Conference on Advanced Information Networking and Applications, AINA Workshops 2020.
‡ All authors contributed equally to this work.

Received: 29 April 2020; Accepted: 31 May 2020; Published: 5 June 2020

Abstract: The Android platform is currently targeted by malicious writers, continuously focused on the development of new types of attacks to extract sensitive and private information from our mobile devices. In this landscape, one recent trend is represented by the collusion attack. In a nutshell this attack requires that two or more applications are installed to perpetrate the malicious behaviour that is split in more than one single application: for this reason anti-malware are not able to detect this attack, considering that they analyze just one application at a time and that the single colluding application does not exhibit any malicious action. In this paper an approach exploiting model checking is proposed to automatically detect whether two applications exhibit the ability to perform a collusion through the *SharedPreferences* communication mechanism. We formulate a series of temporal logic formulae to detect the collusion attack from a model obtained by automatically selecting the classes candidate for the collusion, obtained by two heuristics we propose. Experimental results demonstrate that the proposed approach is promising in collusion application detection: as a matter of fact an accuracy equal to 0.99 is obtained by evaluating 993 Android applications.

Keywords: colluding; malware; model checking; formal methods; security; Android; mobile

1. Introduction

Malware (contraction word for *mal*icious soft*ware*) is currently afflicting each kind of device equipped with an operating system, from workstations to our mobile devices (for instance, smartphone and tables). The final aim of malware is generally to exfiltrate the private and sensible information stored on these devices, typically with an always-on internet connection. According to CLUSIT 2020 security report, 1670 cyber attacks were carried out in 2019, with a growth rate of 7.6% on 2018 and 91.2% compared to 2014. Cyber crime is the main cause of attack, among the means the most used is malware https://clusit.it/rapporto-clusit/.

In this scenario, our mobile devices have become in few years a really appealing surface attack for malware writers, considering the plethora of private and sensitive information that they keep.

Just as example, in modern mobile devices the users can easily download applications from the official market but also from non official ones. These last can be untrustworthy and represent a serious threat for the users' data, in fact users usually consider third-party markets to find free versions of applications that are usually paid ones on the official market [1].

As a confirmation of this trend, researchers shown that several AppChina http://www.appchina.com/ (a widespread Chinese third-party market) applications are supplying malware-tainted applications that can steal data or cost money by auto-subscribing infected users to SMS/MMS services https://www.techinasia.com/china-android-app-stores-malware. But even the Android official market (i.e., Google Play) is not free from infected applications [2]. As a matter of fact, BitDefender security researchers recently (January, 2020) have discovered the presence of 17 applications infected with malware on the Google store, which mostly infested smartphones with aggressive advertisements that showed up even when the applications were not running https://www.ilfattoquotidiano.it/2020/01/16/google-play-store-trovate-altre-17-app-infettate-da-malware/5674489/.

The mobile operating systems most afflicted by malware is Android https://gs.statcounter.com/os-market-share/mobile/worldwide: this is not surprising considering that is currently the most diffused operating system for mobile devices [3]. Moreover, being open, this system is very attracting for malware writers, for these reasons it represents the primary target of cybercriminals, that are able to develop malicious code to attack the users and their information [4,5]. Moreover, also ransomware is becoming a serious threat in Android environment [6,7].

If on the one side we have the attackers, there is also the defensive counterpart (the defenders). With the word "defender" we refer to tools like the anti-malware, but they are not able to identify new malware or threats, because the identification is possible only if the malicious payload signature is stored in the anti-malware's repository [8]. For this reason a threat can be detected only if its signature is present in the anti-malware repository.

One of the new threats appeared in mobile environment is represented by the so-called *collusion* attack. With the collusion attack the malicious action is split between different applications and these communicate with each other when is performed a specific action by the users or when a specific system event occurs. In this way the user has not suspicious because the applications require the minimum permissions needed to launch the attack.

To perform a collusion attack, we need to have at least two applications. To better understand how it works we provide an example: during a collusion one application might read sensitive data (this implies that it has only the permission to read the data) and transmits it to the other application, this one then send the data to the outside world (this implies instead that it has connection permission) [9]. Analyzing separately the applications involved, it is not possible to intercept this kind of attack [10].

It is necessary to underline that in Android system the applications are not completely independent to each other, but they can use the Inter-Component Communication (ICC), a mechanism that allows functionality reuse to reduce the developers' burden. This inter-application collaboration is possible thanks to information exchange between components that could belong to the same application or to different applications [11]. Unfortunately, the ICC model can be misused by malicious applications to threaten user privacy [9,10].

Our research work starts from just described considerations and in this paper we present a tool developed starting from an approach based on model checking, able to detect the collusion between Android applications [12]. We also propose *two* heuristic functions with the aim to reduce the number of the analyzed applications. The functions has been built using the μ-calculus temporal logic. During this work we have focused the attention on *String*, *Int* and *Float* resources shared, exploiting Android *SharedPreferences*.

This work represents an extension of the preliminary paper entitled "Colluding Android Apps Detection via Model Checking" [13] presented at the "34th International Conference on Advanced Information Networking and Applications, AINA Workshops 2020". With respect to the previous work, in this paper we introduce following contributions: we deep explain the formal model we considered to detect collusion; we present the temporal logic formulae to detect *SharedPreferences* sharing *Int* and *Float* values; we show the temporal logic formulae aimed to detect the information (for instance the IMEI or the location of the device) shared through *SharedPreferences*; we show a running example to better understand the proposed approach; we better evaluate the proposed approach by

adding in the experiment a set of 100 malicious and 100 legitimate real-world applications; finally we compare the proposed approach with the current state-of-the-art literature in terms of kind of approach, components handled, inter-application analysis and accuracy. Moreover 20 colluding application developed by authors are considered in the experiment, for a total of 993 evaluated applications.

The paper's organization is the following: in Section 2 we describe the proposed approach for detecting colluding attack in Android environment, a running example is provided in Section 3, evaluation results are discussed in Section 4 in Section 5 current state-of-the-art literature is analyzed and, finally, conclusion and future research lines are drawn in Section 6

2. Detecting SharedPreferences Collusion in Android Environment

The developed approach for the detection and verification of Android colluding applications starts from the binary analysis code using the Java ByteCode, because the source code is not always reachable like the binary code [14,15].

Figure 1 shows the schema of the proposed approach.

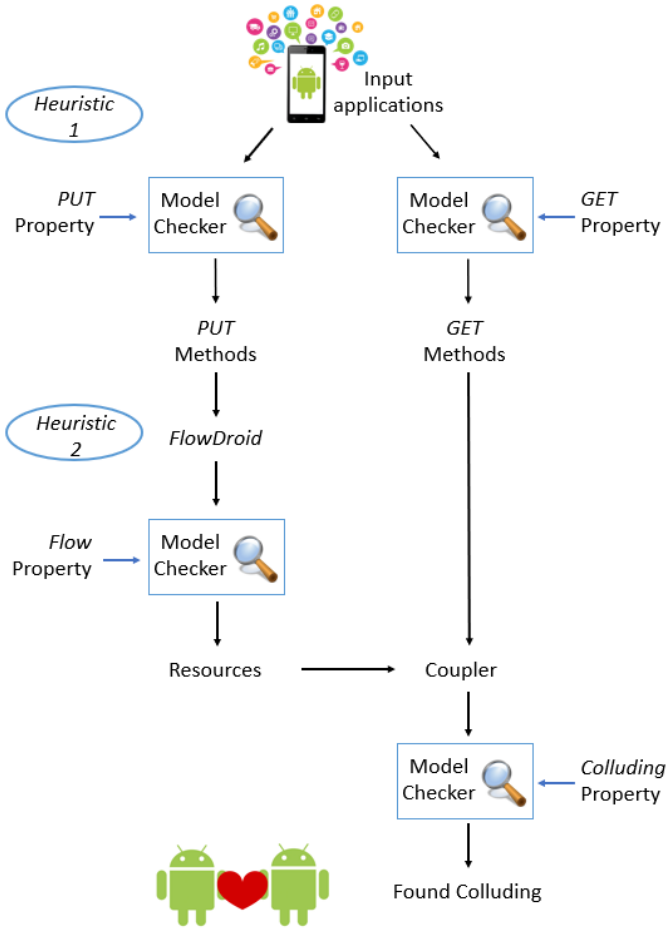

Figure 1. The proposed approach for colluding application detection. We highlight the First Heuristic (with the φ_{PUT} and the φ_{GET} properties) and the Second Heuristic (with the χ_P property) aimed to reduce the application comparisons.

In this paper we experiment whether model checking techniques can be adopted for detecting Android collusion. To apply model checking we need a model and a set of properties.

For this reason, in the first step we have defined a formal model from the Java Bytecode of an Android application. With this technique it is possible to build a general model and in this model we can check the system properties.

The formal model is expressed in the Calculus of Communicating Systems (CCS) process calculus, while the properties are expressed in μ-calculus. The proposed formal model represents a process that starting from the ByteCode is able to simulate the application behaviour. The *.apk* (i.e., Android Package) is an Android application and is a variant of the .jar file compiled for embedded devices. This type of file contains the executable code (i.e., the *.dex* file) targeting the Dalvik Virtual Machine, the resource folder (i.e., images, sounds, icons) and the Manifest file.

To obtain the Java ByteCode from the *.apk* file, we need:

- to use the dex2jar https://sourceforge.net/projects/dex2jar/ tool, to generate the *.jar* file starting from the *.apk* file;
- to use the Java Archive Tool utility to extract from the *.jar* file the java classes and packages https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jar.html;
- to invoke the BCEL (Byte Code Engineering Library https://commons.apache.org/proper/commons-bcel/) to generate the Android application ByteCode from the *.jar* file.

After this, is executed an algorithm developed by the author in [2,3] to generate a CCS process for each Java ByteCode instructions. The process to generate the CSS models is aimed to codify the instructions: an action of the model represents a single Java Bytecode instruction.

2.1. The First Heuristic Function: PUT and GET

Given the large number of applications present in official and unofficial stores, the cost of the analysis grows exponentially with the number of applications analyzed at the same time. To reduce the number of colluding candidates to test, we need to find groups of applications that should be considered together for collusion.

We have defined some temporal logic formulae to detect the applications behaviour and recognize collusive ones. The first heuristic function works using the *SharedPreferences*. It checks every read or write of a *String*, *Int* and *Float* shared resources in every possible code path. So, is applied a division of the applications relatively to the different shared resource use.

Below we show two Android source code snippets (belonging to two different Android applications), the first one represents a *SharedPreferences GET* (i.e., a *read* operation) of the *String* value (i.e., "defaultValue", read from the "value1" variable).

```
//GET snippet: start
SharedPreferences sharedPreferences = this.getSharedPreferences("SharedPreferences",
Context.MODE_WORLD_WRITEABLE);
String value1 = sharedPreferences.getString("value1","defaultValue");
//GET snippet: end
```

The second snippet, below shown, represents a *SharedPreferences PUT* (i.e., a *write* operation), where the "information" value is stored in the "value1" variable.

```
//PUT snippet: start
SharedPreferences sharedPreferences= this.getSharedPreference
("SharedPreferences", Context.MODE_WORLD_WRITEABLE);
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.putString("value1", "information");
//PUT snippet: end
```

The *GET* and *PUT* snippets show how it is possible for two Android applications to share (sensitive) data between two applications without requiring additional permissions.

As shown from the source code snippets, an application can execute two different operations on a shared resource: *PUT* and *GET*. These actions can be encoded with the μ-calculus logic:

- when an application executes a *PUT* action on a shared resource, the formula (Table 1—*Formula*1) results true if are performed the following actions: *invokegetSharedPreferences*, *invokeedit*, *invokeputString*/*invokeputInt*/*invokeputFloat*, *invokecommit*;
- when instead an application executes a *GET* action on a shared resource, the formula (Table 1—*Formula*2) results true if are performed the following actions: *invokegetSharedPreferences*, *invokegetString*/*invokegetInt*/*invokegetFloat*.

Table 1. The First Heuristic: the φ_{PUT} property is aimed to detect methods invoking *PUT* operations on *SharedPreferences*, while the φ_{GET} property is aimed to detect methods invoking *GET* operations on *SharedPreferences*.

*Formula*1		
φ_{PUT}	$=$	$\mu X.\langle invokegetSharedPreferences\rangle\ \varphi_{PUT_1} \vee$ $\langle -invokegetSharedPreferences\rangle\ X$
φ_{PUT_1}	$=$	$\mu X.\langle invokeedit\rangle\ \varphi_{PUT_2} \vee \langle -invokeedit\rangle\ X$
φ_{PUT_2}	$=$	$\mu X.\langle invokeputString, invokeputInt, invokeputFloat\rangle\ \varphi_{PUT_3} \vee$ $\langle -invokeputString, invokeputInt, invokeputFloat\rangle\ X$
φ_{PUT_3}	$=$	$\mu X.\langle invokecommit\rangle\ \mathtt{tt} \vee \langle -invokecommit\rangle\ X$
*Formula*2		
φ_{GET}	$=$	$\mu X.\langle invokegetSharedPreferences\rangle\ \varphi_{GET_1} \vee$ $\langle -invokegetSharedPreferences\rangle\ X$
φ_{GET_1}	$=$	$\mu X.\langle invokegetString, invokegetInt, invokegetFloat\rangle\ \mathtt{tt} \vee$ $\langle -invokegetString, invokegetInt, invokegetFloat\rangle\ X$

The heuristic function allows to obtain in a short time two different sets of applications to be analyzed later, where in the first one are contained the applications that verify the *PUT* property, instead in the second one are contained the applications that verify the *GET* property. The reduction in processing costs is given thanks to model checking (which considers as input a class modeled in terms of CCS and a temporal logic formula) which performs a screening and selects only the classes that could potentially generate a collusion. In this way the number of classes to be tested is significantly reduced.

2.2. The Second Heuristic Function: FlowDroid

We have thought to a second heuristic based on the flow analysis, with the aim to further reduce the search space of colluding applications. If a *PUT* formula is verified during execution, will be executed *FlowDroid* to check the data flow presence, but it will not be executed if a *GET* property is verified.

This choice is made because a *PUT* action presupposes the presence of an active modification and the analysis of the flow can help us understand if some modification has been made.

In Table 2 are showed the properties used to verify data flow on the model generated from the *FlowDroid* output. The properties are useful to classify the flow type according to the channel where are passed the data to a specific shared resource.

The channel in this case is represented by the exfiltrated data: we consider *WIFI* (detected by the ψ_{WIFI} formula in Table 2), *IMEI* (detected by the ψ_{IMEI} formula in Table 2), *GPS* (detected by the ψ_{GPS} formula in Table 2), *ACCOUNTS* (detected by the $\psi_{ACCOUNTS}$ formula in Table 2), *TASK* (detected by the ψ_{TASK} formula in Table 2), *CONT_BOOK_HIST* (detected by the ψ_{CONT} formula in Table 2) and *CALL* (detected by the ψ_{CALL} formula in Table 2).

Table 2. The Second Heuristic, aimed to check with the χ_P property the resource (for instance the device IMEI or the accounts) shared between the *SharedPreferences* on the model built exploiting the FlowDroid tool.

*Formula*3	
χ_1	$= \nu X.[query]\,\texttt{ff} \wedge [-query]X$
χ_2	$= \nu X.[getString]\,\texttt{ff} \wedge [-getString]X$
ψ_{CALL_1}	$= \mu X.\langle getString\rangle\, \psi_{CALL_2} \vee \langle -getString\rangle\, X$
ψ_{CALL_2}	$= \mu X.\langle putString\rangle\, \texttt{tt} \vee \langle -putString\rangle\, X$
ψ_{CALL}	$= \psi_{CALL_1} \wedge \chi_1$
χ_3	$= \mu X.\langle getString\rangle\, \texttt{tt} \vee \langle -getString\rangle\, X$
ψ_{CONT_1}	$= \mu X.\langle query\rangle\, \psi_{CONT_2} \vee \langle -query\rangle\, X$
ψ_{CONT_2}	$= \mu X.\langle putString\rangle\, \texttt{tt} \vee \langle -putString\rangle\, X$
ψ_{CONT}	$= \psi_{CONT_1} \wedge \chi_3$
ψ_{TASK}	$= \mu X.\langle getRunningTasks\rangle\, \psi_{TASK_1} \vee \langle -getRunningTasks\rangle\, X$
ψ_{TASK_1}	$= \mu X.\langle putString\rangle\, \texttt{tt} \vee \langle -putString\rangle\, X$
$\psi_{ACCOUNTS}$	$= \mu X.\langle getAccounts\rangle\, \psi_{ACCOUNTS_1} \vee \langle -getAccounts\rangle\, X$
$\psi_{ACCOUNTS_1}$	$= \mu X.\langle putString\rangle\, \texttt{tt} \vee \langle -putString\rangle\, X$
ψ_{GPS}	$= \mu X.\langle getLastKnownLocation\rangle\, \psi_{GPS_1} \vee \langle -getLastKnownLocation\rangle\, X$
ψ_{GPS_1}	$= \mu X.\langle putString\rangle\, \texttt{tt} \vee \langle -putString\rangle\, X$
ψ_{IMEI}	$= \mu X.\langle getSimSerialNumber, getDeviceId\rangle\, \psi_{IMEI_1} \vee \langle -getSimSerialNumber, getDeviceId\rangle\, X$
ψ_{IMEI_1}	$= \mu X.\langle putString\rangle\, \texttt{tt} \vee \langle -putString\rangle\, X$
χ_{TASK}	$= \nu X.[getRunningTasks]\,\texttt{ff} \wedge [-getRunningTasks]X$
$\chi_{ACCOUNTS}$	$= \nu X.[getAccounts]\,\texttt{ff} \wedge [-getAccounts]X$
χ_{GPS}	$= \nu X.[getLastKnownLocation]\,\texttt{ff} \wedge [-getLastKnownLocation]X$
χ_{SIM}	$= \nu X.[getSimSerialNumber]\,\texttt{ff} \wedge [-getSimSerialNumber]X$
χ_{ID}	$= \nu X.[getDeviceId]\,\texttt{ff} \wedge [-getDeviceId]X$
χ_{IMEI}	$= \chi_{SIM} \vee \chi_{ID}$
$\psi_{WIFI-AND}$	$= \chi_{TASK} \wedge \chi_{ACCOUNTS} \wedge \chi_{GPS} \wedge \chi_{IMEI}$
ψ_{WIFI_1}	$= \mu X.\langle toString\rangle\, \psi_{WIFI_2} \vee \langle -toString\rangle\, X$
ψ_{WIFI_2}	$= \mu X.\langle putString\rangle\, \texttt{tt} \vee \langle -putString\rangle\, X$
ψ_{WIFI}	$= \chi_2 \wedge \chi_1 \wedge \psi_{WIFI-AND} \wedge \psi_{WIFI_1}$
χ_P	$= \langle \psi_{WIFI} \vee \psi_{IMEI} \vee \psi_{GPS} \vee \psi_{ACCOUNTS} \vee \psi_{TASK} \vee \psi_{CONT} \vee \psi_{CALL} \rangle$

2.3. The Formal Model Design and Generation

We have create a system model and system properties to be able to use the model verification techniques. For the model development, we start with the definition of an algorithm that is based on the use of *FlowDroid* to take as input one application at a time and to obtain in output an XML file containing the data flow reconstruction that includes the description of the sources and (respective) sinks. Subsequently will be created two lists, one for the sources and one for the sinks respectively. The two lists are used to compose the hashmap, where the key is represented by the source and the value is represented by the array sink.

The model is created building a graph based on the flow, that is composed by an array of roots and an array of leaves. During the execution, the sources array is scanned to verify if the source element is present also in the sinks array and if it is true, then the element is deleted from both arrays. At the end we will find in the sources array all the roots and in the sinks array all the leaves of the flow graph.

We developed a recursive algorithm working in the following way: it selects the first root contained in the roots array, to enter into the hashmap using the root like a key and checks if it has sons. In case there is a son, in the CCS file model we go from the root to the son and the recursive

algorithm will be invoked until all the roots have been selected. In this way, we obtain the CCS file for the model checking.

2.4. Coupling Process and Formal Verification

With the heuristic functions application, we are able to obtain a set of applications pairs for the *SharedPreferences* shared resources: these resources can be *String*, *Int* of *Float*. We indicate these resources as $S_{s.r.}$.

For each $(p, q) \in S_{s.r.}$ the CCS process is defined as:

$$Proc_{pq} = (p \mid C \mid q) \backslash L$$

To better understand:

- p and q are the CCS representation of potential candidates to the collusion;
- C is the definition of the coupling process that verify the collusion presence among p and q;
- L is the set of communication actions and is composed how showed:

$L = \{Preferences\_NAME,\ invokegetSharedPreferences,\ resource\_ID,$
$invokeputString/\ invokeputInt/\ invokeputFloat,\ invokegetString/\ invokegetInt/\ invokegetFloat\}$.

At the end is applied a formal verification environment including a model checker. For each $(p, q) \in S_{s.r.}$ is checked the CCS process $Proc_{pq}$. The model checker tool used is CWB-NC, when its result is true, then means CCS process $Proc_{pq}$ satisfies a mu-calculus formula encoding the colluding notion. So we can deduce that our approach under colluding analysis considers two applications p and q, false otherwise.

In this paper we experiment with the Concurrency Workbench of New Century (*CWB-NC*) https://sourceforge.net/projects/cwb-nc/ model checker, a tool considering a plethora of different techniques for specifying and verifying finite-state concurrent systems.

3. Android Colluding Detection Running Example

To better understand how the proposed approach is working to malicious colluding detection, in following section we show a running example. We provide the several model automatically generated by the proposed approach. Let us consider two applications: the first one is identified by *FMVPNMY3DT87FD4A2D15ON3KR0243ZUH* hash and the second one by the *B5KH2TM08LBYPA03KHFOWA3E7JB9W0IU* hash.

As highlighted from Figure 1 the first step is represented by the *PUT* and the *GET* properties verification. Thus the proposed approach converts all the classes (parsed in ByteCode format obtained by exploiting the BCEL library) into CCS processes. So, each CCS process is checked respectively with the *PUT* and the *GET* formulae.

Figures 2 and 3 shows the graph snippets obtained from the CCS processes that the CWB-NC model checker respectively labelled as *true* when the *GET* and the *PUT* formulae are verified.

The images in Figures 2 and 3 are given by the representation of an application in a call graph structure, where each arch identifies a Java ByteCode instruction and it is possible to obtain it thanks to the transformation of the class into a CCS process, that simulates the applications' beaviour. In detail we obtained the call graph structure by invoking in the first time "load file.ccs" command on the CWB-NC and then "compile automaton-name.ccs". The two commands respectively permit to load the file in the CWB-NC model checker, while the second enable us to obtain the graph of the automaton.

Each connection in the call graph generated from the model checker is in the following format: *start* : *action* {*end*} where, *start* : represents the source node, *action* the ByteCode instruction (i.e., the node action) and with *end* the destination(s) node(s).

Figure 2 shows the call graph related to CCS process resulting *true* to the *GET* formula.

```
21:     load    {23}

22:     push7LHWDB30YDZG7KQ     {24}

23:     pushnotfound    {25}

24:     push    {26}

25:     invokeequals    {27}

26:     invokegetSharedPreferences      {28}

27:     ifeqff  {29}
        ifeqtt  {30}

28:     push18N2C3EFRT0L51O     {31}

29:     push5000        {32}

30:     load    {33}

31:     pushnotfound    {34}

32:     invokesleep     {30}

33:     pushnotfound    {35}

34:     invokegetString {36}

35:     invokeequals    {37}

36:     store   {21}
```

Figure 2. The GET automaton: the *invokegetSharedPreferences* and *invokegetString* actions are related to a read operation (in this case of a *string* variable) from a *SharedPreferences*.

According to φ_{GET} formula shown in Table 1, the model checker output true when the φ_{GET} formula is checked because it exists a sequence of following instructions: *invokegetSharedPreferences*, *invokegetString*. In particular in Figure 2 the *invokegetSharedPreferences* instruction is the label between the 26 and the 28 nodes, while the *invokegetString* instruction is the label between the 34 and the 36 nodes.

In Figure 3 we show a snippet related to the call graph obtained from the *PUT* CCS process.

According to φ_{PUT} formula shown in Table 1, the formal verification environment outputs *true* on the graph shown in Figure 3 because it contains following path: *'invokegetSharedPreferences*, *'invokeedit*, *'invokeputString* and *'invokecommit*. From Figure 3 it emerges that the *'invokegetSharedPreferences* instruction is the label between the 13 and the 15 nodes, the *'invokeedit* instruction is the label between the 15 and 17 nodes, *'invokeputString* is the label between the 20 and 21 nodes and, finally, *'invokecommit* is the label between nodes 21 and 22.

Once the φ_{PUT} formula shown in Table 1 is satisfied on a CCS process, the *FlowDroid* tool is invoked to generate a CCS process aimed to understand the kind variable that is wrote into the *SharedPreferences*.

In this example, the automaton generated from the *FlowDroid* output is shown in Figure 4: whether the property shown in Table 2 is satisfied, one of this data is written into the *SharedPreferences*.

```
8:      load    {10}

9:      'push7LHWDB30YDZG7KQ    {11}

10:     t       {12}

11:     push    {13}

12:     load    {14}

13:     'invokegetSharedPreferences     {15}

14:     'invokeinit     {16}

15:     'invokeedit     {17}

16:     return  {18}

17:     'push18N2C3EFRT0L51O    {19}

18:
19:     load    {20}

20:     'invokeputString        {21}

21:     'invokecommit   {22}

22:     pop     {23}

23:     'pushCollusion  {24}

24:     'pushFilesavedtoSharedPreferences       {25}

25:     'invokev        {26}
```

Figure 3. The PUT automaton: the *'invokegetSharedPreferences*, *'invokeedit* and *'invokeputString* actions are related to a write operation (in this case of a *string* variable) from a *SharedPreferences*.

```
cwb-nc> compile ALL
Building automaton...
States: 8
Transitions: 13
Done building automaton.
0:      getString       {1, 2}

1:      toString        {3}

2:      toString        {3}

3:      init    {4}

4:      init    {4}
        putString       {5}
        getSharedPreferences    {6}
        edit    {7}
        commit  {6}
        v       {6}

5:      commit  {6}

6:
7:      putString       {5}


Start States:  [0]
```

Figure 4. The FlowDroid automaton. From the *getString*, *putString* and *getSharedPreferences* actions is emerging that the (read and write) shared variable is a *string*.

In the example depicted in Figure 4, the χ_P property (shown in Table 2) is satisfied because between the several properties concatenated by the *OR* operator the ψ_{CALL} property is satisfied. In fact, in the graph in Figure 4 exists one path with the *getString*, the *putString* actions and without the *query* actions. In particular the *getString* action is the label between the nodes 0 and 1, 2 and the *putString* action is the label between the 4 and 5 nodes.

Figure 5 shows one of the *C* processes generated for the collusion detection between the *PUT* and the *GET* processes with the set of *L* communication actions, as described in Section 2.

```
cwb-nc> compile COUPLER_push7LHWDB30YDZG7KQ_push18N2C3EFRT0L51O
Building automaton...
States: 18
Transitions: 18
Done building automaton.
0:      push7LHWDB30YDZG7KQ     {1}
        push18N2C3EFRT0L51O     {2}

1:      invokegetSharedPreferences      {3}

2:      invokegetSharedPreferences      {4}

3:      push18N2C3EFRT0L51O     {5}

4:      push7LHWDB30YDZG7KQ     {6}

5:      invokeputString {7}

6:      invokeputString {8}

7:      'push7LHWDB30YDZG7KQ    {9}

8:      'push18N2C3EFRT0L51O    {10}

9:      'invokegetSharedPreferences     {11}

10:     'invokegetSharedPreferences     {12}

11:     'push18N2C3EFRT0L51O    {13}

12:     'push7LHWDB30YDZG7KQ    {14}

13:     'invokegetString        {15}

14:     'invokegetString        {16}

15:     COLLUDING_OK_PUSH7LHWDB30YDZG7KQ_PUSH18N2C3EFRT0L51O    {17}

16:     COLLUDING_OK_PUSH18N2C3EFRT0L51O_PUSH7LHWDB30YDZG7KQ    {17}

17:

Start States:  [0]
Execution time (user,system,gc,real):(0.025,0.000,0.000,0.025)
```

Figure 5. The Coupler automaton, in this case the *push7LHWDB30YDZG7KQ* and the *push18N2C3EFRT0L51O* actions (obtained from the *GET* automaton) and the *'push7LHWDB30YDZG7KQ*, and *'push18N2C3EFRT0L51O* actions (obtained from the *PUT* automaton) are symptomatic of a colluding between the *GET* and the *PUT* automata.

The idea behind the *C* automaton is to verify whether the *PUT* and the *GET* automata exhibits the same *push* labels (in other words whether the *PUT* and the *GET* are invoking the *same SharedPreferences* and they are respectively writing and reading the *same* item from the *SharedPreferences*). In this case the *C* automaton contains both the *push7LHWDB30YDZG7KQ* and the *push18N2C3EFRT0L51O* (retrieved from the *GET* model) and the *'push7LHWDB30YDZG7KQ* and the *'push18N2C3EFRT0L51O* obtained from the *PUT* one: in this specific case *18N2C3EFRT0L51O* is the name of the *SharedPreferences* and *7LHWDB30YDZG7KQ* is the key of the value that is wrote by the *PUT* class and read by the *GET* class. Once obtained the *C* process, we generate the Tester process as shown in Figure 6.

```
proc TESTER = (COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 |
COUPLER_push7LHWDB30YDZG7KQ_push18N2C3EFRT0L51O | COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \
{push7LHWDB30YDZG7KQ,push18N2C3EFRT0L51O} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 | COUPLER_push7LHWDB30YDZG7KQ_pushCollusion |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push7LHWDB30YDZG7KQ,pushCollusion} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 |
COUPLER_push7LHWDB30YDZG7KQ_pushFilesavedtoSharedPreferences |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push7LHWDB30YDZG7KQ,pushFilesavedtoSharedPreferences} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 | COUPLER_push18N2C3EFRT0L51O_pushCollusion |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push18N2C3EFRT0L51O,pushCollusion} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 |
COUPLER_push18N2C3EFRT0L51O_pushFilesavedtoSharedPreferences |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push18N2C3EFRT0L51O,pushFilesavedtoSharedPreferences} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 | COUPLER_pushCollusion_pushFilesavedtoSharedPreferences |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {pushCollusion,pushFilesavedtoSharedPreferences}
```

Figure 6. The Tester process, aimed to verify if at least on the Coupler processes exhibits a collusion.

The aim of the Tester process is to verify whether one of the *C* process is related to a collusion. The results of the tool from this analysis (stored in a *.csv* file from the tool) are shown in Table 3.

Table 3. Running example result, with the detail about the name, the class, the method and the variable exhibiting the *GET* and the *PUT* collusion.

Description	Item
GET Application	FMVPNMY3DT87FD4A2D15ON3KR0243ZUH.apk
GET Class	com.acid.tFMVPNMY3DT87FD4A2D15ON3KR0243ZUH.PLAINACTIVITY$1
GET Method	public void run
GET Variable	PUSH7LHWDB30YDZG7KQ
PUT Application	B5KH2TM08LBYPA03KHFOWA3E7JB9W0IU.apk
PUT Class	com.acid.rB5KH2TM08LBYPA03KHFOWA3E7JB9W0IU.PlainActivity$1
PUT Method	public void run
PUT Variable	PUSH18N2C3EFRT0L51O

As appears from the results in Table 3, the proposed approach is able to detect the name of the *GET* and the *PUT* applications, the package, the class and the method where the *GET* and *PUT* collusion action occurs, the name of the *SharedPreferences* involved in the attack and the key of the value shared by the collusion applications.

4. Experimental Evaluation

In this section we present the experimental results obtained by the proposed approach. We experiment a real-world data-set composed by malware with a standalone malicious payload (i.e., no requiring another application to perpetrate the malicious behaviour), colluding malware and trusted applications. Below we first present the data-set involved in the experiment and then the results. Moreover, we discuss the reduction rate exhibited by the proposed approach in term of application comparison.

4.1. The Data-Set

Several sources are used to build the data-set to evaluated our approach. In particular we consider:

- the *ACE* data-set, proposed by authors in [16]. In this paper authors have developed a system to automatically generate combinations of colluding (and non-colluding) applications in Android environment, the so-called *Application Collusion Engine (ACE)*, with the aim to evaluate different collusion detection and protection methods. About the experiment we have used a data-set generated with ACE, we have generated 160 different applications in order to have 80 couples of colluding applications that communicate through *SharedPreferences* with *String* variables and other 320 different applications in order to have 160 couples of colluding applications not using this type of communication;
- a set of 20 applications developed by authors (i.e., the *SP_INT_FLOAT* data-set). In this data-set 10 applications are performing a collusion attacks through an *Int* value, while the

other 10 applications consider the collusion attack through a *Float* value. We built this data-set to evaluate the proposed approach on the collusion through *Int* and *Float* variables (in fact, the *ACE* data-set considers only collusion through *String* variables);

- the *DroidBench* 2.0 https://github.com/secure-software-engineering/DroidBench composed by 119 applications. We are talking about an open data-set with the aim to evaluate the effectiveness of taint-analysis tools for Android applications. This data-set do not have colluding applications, but some applications use *SharedPreferences*;
- the *Swansea* University data-set, composed by 14 applications. In it there are only two colluding applications;
- the *Drebin* malware repository, a data-set of well-known malware [17,18] not performing collusion attacks, with the aim to demonstrate the ability of the proposed approach to detect only colluding malware. We select 10 malware belonging to the top 10 malicious family in the data-set for a total of 100 not colluding malware, as shown in Table 4;

Table 4. Malware families in the *Drebin* data-set. We selected the 10 most populous families in this repository, for a total of 100 not colluding malware. We indicate in the *Inst.* column the payload delivering (standalone, repackaging, update), in the *attack* column the kind of attack (trojan, botnet) and in the *Activation* column the operating system events triggering the malicious payload.

Family	Inst.	Attack	Activation
FakeInstaller	s	t,b	
DroidKungFu	r	t	boot,batt,sys
Plankton	s,u	t,b	
Opfake	r	t	
GinMaster	r	t	boot
BaseBridge	r,u	t	boot,sms,net,batt
Kmin	s	t	boot
Geinimi	r	t	boot,sms
Adrd	r	t	net,call
DroidDream	r	b	main

- a set of legitimate 260 applications obtained from the official store of Google i.e., *Play* store. These applications were automatically collected from Google Play (https://play.google.com/store), by using a script developed by authors aimed to query and to download applications from Android official market. In order to confirm the trustworthiness of these 260 applications we analyzed this data-set by exploiting the VirusTotal service (https://www.virustotal.com/). This service run 60 different antivirus software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others) on each application: the output confirmed that the trusted applications included in the legitimate data-set did not contain malicious payload.

By summarize, we consider a total of 993 applications. In the data-set there are colluding applications, malware application not performing the collusion attack and legitimate applications obtained from different sources.

4.2. Collusion Detection Results

In this section we present the evaluation results.

In order to assess the performance of the proposed approach for colluding detection, we consider four different metrics: Precision, Recall, F-Measure and Accuracy.

The precision has been computed as the proportion of the examples that truly belong to class X among all those which were assigned to the class. It is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved:

$$Precision = \frac{tp}{tp + fp}$$

where *tp* indicates the number of true positives and *fp* indicates the number of false positives.

The recall has been computed as the proportion of examples that were assigned to class X, among all the examples that truly belong to the class, i.e., how much part of the class was captured. It is the ratio of the number of relevant records retrieved to the total number of relevant records:

$$Recall = \frac{tp}{tp + fn}$$

where *fn* indicates the number of false negatives.

The F-Measure is a measure of a test's accuracy. This score can be interpreted as a weighted average of the precision and recall:

$$F\text{-}Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

The Accuracy is the fraction of the classifications that are correct and it is computed as the sum of true positives and negatives divided all the evaluated samples:

$$Accuracy = \frac{tp + tn}{tp + fn + fp + tn}$$

where *tn* indicates the number of true negatives.

In Table 5 we show the obtained results.

Table 5. Performance Results. The proposed approach reaches a precision and a recall equal to 0.98 with 1 false positive and 1 false negative.

TP	TN	FP	FN	PR	RC	Fm	AC
90	160,779	1	1	0.98	0.98	0.98	0.99

As shown by the results in Table 5 we obtain an accuracy equal to 0.99. We obtain only 2 misclassifications in particular, 1 false positive (one application belonging to the *DroidBench* data-set) and 1 false negative (the colluding application belonging to the *Swansea* data-set).

To demonstrate how the proposed heuristics are effective to reduce the comparison between applications, in Table 6 we show the reduction rate obtained by the proposed approach.

Table 6. Reduction rate results. Without the proposed heuristics the checking for collusion of 993 requires 160,882 comparisons, while with the proposed heuristics we need 90 comparisons for colluding detection.

Data-Set	Applications	Theoretical Couples	Not Colluding Couples	Colluding Couples
ACE	480	114,960	114,880	80
SP_INT_FLOAT	20	190	180	10
DroidBench	119	7021	7021	0
Swansea	14	91	91	0
Drebin	100	4950	4950	0
Play	260	33,670	33,670	0
TOTAL	993	160,882	160,781	90

As shown from reduction rates in Table 6, the evaluation of 993 applications require 160,882 theoretical couples: the proposed approach for detecting collusion attacks considers only 90 couples (with a reduction rate equal to 99.94%).

5. Colluding and Malicious Communication Channel Detection State-of-the-Art

In the detection techniques environment usually the attention is focused on the analysis of one sample at a time, not considering the communication channel that can be used by cybercriminals to perform the attack. This section reviews the related work that consider methods focused on the detection of colluding and communication channel.

For instance, in [19] the researchers have developed a tool that uses static taint analysis technique with the aim to retrieve paths in which private and sensitive information are sent to external without ask the users consent. Using IccTA is possible to detect paths with a single component or multiple components. The researchers have developed about 22 applications containing ICC-based privacy leaks to verify this approach.

Amandroid [20] is an approach developed starting from the aforementioned IccTA that focuses it attention on leaks detection with an ICC analysis. To execute the ICC analysis, it needs to generate two components: the Inter-Component Data Flow Graph (ICDFG) and the Data Dependence Graph (DDG). It executes data flow and data dependence analysis for the components and tracks communication activities between the latter, allowing a customized analysis on Android applications.

About the colluding attack with covert channels, the authors in [21] have developed a multichannel communication mechanism to transfer sensitive data in secure way on mobile devices. This mechanism is called Multichannel Communication System (MSYM) and uses the Android VpnService interface (https://developer.android.com/reference/android/net/VpnService) to intercept the network data sent, after this, splits it into different parts that will be disordered and encrypted via multiple transmission channels.

The accelerometer sensor can generate signals able to reflect the motions of the users, so it is possible use it like covert channel to exchange data. Using properly the devices' vibration motor, it is possible to encode the data to avoid that malicious applications read or steal them. With this mechanism an application can produce effects on acceleration data to be received and decoded by a second application. In [22] are been implemented two Android applications that play the roles of source and sink, furthermore are used three different smartphones models to verify the communication.

XManDroid [23] provides runtime monitoring of communication links between applications and defines communication classification based on permission policies. This tool presents a very high number of false positive (55%). The aim is to minimize the risk of applications collusion using different security metrics.

IIFA (Modular Inter-Application Intent Information Flow Analysis of Android Applications) [24] is an approach based on an intent-flow pre-analysis, it avoids the combinatorial explosion between all the communication partners, furthermore excludes the infeasible communication paths.

DIALDroid [25] performs a systematic large-scale security analysis on ICC-based sensitive inter-application data flows. It uses relational database to match ICC entry and exit points.

The authors in [26] show the ability to send data using the USB (Universal Serial Bus) which acts like a covert channel to the public charging station. They implemented an application called PowerSnitch, that is able to send data through power bursts, working on the power consumption of the devices' CPU.

MR-Droid [27] is a tool to detect inter-application communication threats in particular intent spoofing and collusion. It uses a framework based on MapReduce to execute a compositional application analysis.

TaintDroid [28] is an approach that represents an Android Operating System extension. Through third-party applications, it tracks sensitive data flow. It starts assuming that applications downloaded by third-parties are not reliable, for this reason it checks in real time how them access and modify these sensitive data.

Researchers in [29] have developed a method to identify colluding applications analyzing communication channels, permissions, access to sensitive data and others features. It is possible to detect collusion in Android environment with it.

IntelliDroid [30] is a tool able to generate input specific for dynamic analysis tool, reducing the false positives and allowing more precise analysis. IntelliDroid allows to execute targeted analysis.

Authors in [31] propose a method aimed to check the Android inter-application communication dynamically. After the activity component analysis of the application, it implements different attack scenarios, considering as vulnerabilities: Cross-Site Scripting, SQL Injection, User Interface Spoofing, File Manipulation, Native Memory Corruption, Unsafe Reflections, Fragment Injection and Java Crashing.

Table 7 shows a comparison between the current literature we discussed and the proposed approach. We grouped the works in terms of *Kind of Approach, Components Handled, Inter-Application Analysis and Accuracy. In detail, in the Kind of Approach column in Table 7*, coherently with authors in [32], we consider methods that do not require the execution of the application (i.e., *Static*), methods requiring the application execution (i.e., *Dynamic*) and methods considering policies for detecting collusion attacks (i.e., *Policy Enforcement*). The *Components Handled* column in Table 7 highlights the colluded shared resources detected: *A* is for *Activity*, *S* is for *Service*, *R* for *Broadcast Receiver*, *C* for *Content Provider* and *SP* for *Shared Preferences*. The *Inter-Application Analysis* column specifies whether the discussed methods consider resources shared between different applications (*Yes*) or consider only resources shared within the same application (*No*), while the *Accuracy* column shows the performances (*n.a.* stands for not available).

Table 7. State-of-the-art comparison in colluding detection.

Research	Year	Kind of Approach	Components Handled	Inter-Application	Accuracy
AmanDroid [20]	2014	Static	A, S, R	No	0.67
DIALDroid [25]	2017	Policy Enforcement	A, S, R, C	Yes	0.91
IccTA [19]	2015	Static	A, S, R, C	Yes	0.93
IntelliDroid [30]	2016	Dynamic	A, S, R	No	0.93
Hay et al. [31]	2015	Dynamic	A	Yes	n.a.
MR-Droid [27]	2017	Static	A, S, R	No	1
TaintDroid [28]	2014	Dynamic	A, S, R, C	No	n.a.
Asavoae et al. [29]	2016	Static	A, S, R	No	0.94
XManDroid [23]	2011	Policy Enforcement	A, S, R, C	Yes	n.a.
Our Approach	2020	Static	SP	Yes	0.99

From the analysis in Table 7 emerges that the proposed approach is the first one (at the best of authors knowledge) considering the *SharedPreferences* as inter-application communication channel for colluding attacks detection. As a matter of fact, the approaches available in the current-state-of-the-art considers inter- and intra-communication between other components (activities, service, broadcast receivers and content providers), this represent a novelty point of the proposed approach, aimed to detect whether a *String*, an *Int* or a *Float* variable is shared exploiting this channel. Furthermore, from the performance of view, the only method overcoming the proposed approach is *MR-Droid* proposed by Liu et al. [27] . Differently from the proposed approach, authors in [27] do not consider *SharedPreferences* as communication channel. Moreover they evaluate in the experimental analysis 8 applications, while we experiment the proposed approach using a data-set (obtained from different repositories and with a set of applications developed by authors) composed by 993 mobile applications.

6. Conclusions and Future Work

Mobile malware writers are continuously increasing the techniques to develop more complex and undetectable malicious payloads, with the aim to elude the current detection mechanism provided by signature-based anti-malware. One of the latest trend in the mobile attack landscape is represented

by the so-called collusion attack. In this paper we propose an approach based on formal methods for detecting this new threat in Android environment. Basically we represent Android applications in term of CCS processes and, by exploiting model checking, we verify whether the automata obtained from the classes of the application satisfy certain properties formulated by the authors. There are several collusion attacks in the wild, in this experiment we focus on the colluding attacks through *SharedPreferences*, object pointing to a file containing key-value pairs providing a simple way to read and write them, really widespread to share data between different Android applications. We propose several heuristics to drastically reduce the number of comparisons performed by the proposed approach. In detail the first heuristic is aimed to looking for methods aimed to *GET* and to *PUT* information on the *SharedPreferences*. The second heuristic is aimed to detect the kind of information exfiltrated exploiting the *SharedPreferences* channel (i.e., *WIFI*, *IMEI*, *GPS*, *ACCOUNTS*, *CONT_BOOK_HIST* and *CALL*).

To evaluate the effectiveness of the proposed approach are considered standalone malicious applications, legitimate applications crawled by the official Android market and a set of applications performing malicious collusion. In particular, we built a data-set by exploiting several repositories freely available for research purposes (i.e., *ACE*, *DroidBench*, *Swansea* and the Google Play Store). Moreover we take into account 20 applications developed by authors exhibiting the *SharedPreferences* collusion with integer and float variables.

Experimental results show the effectiveness of the proposed approach: as a matter of fact an accuracy equal to 0.99 is obtained, obtaining only two application misclassifications. The proposed heuristics are able to obtain a comparison reduction rate of 99.94%, in fact the proposed approach requires only 90 comparisons instead of the 160,882 comparisons required to evaluate the 993 applications in the analyzed data-set. As future work we plan to detect different kinds of collusion attacks for instance, the external storage one. Moreover, we will extend the proposed approach to detect collusion attacks performed by more than two applications. Furthermore, we are investigating whether the adoption of formal equivalence checking can be helpful to obtain better performances on the detection of malicious communication channels between several mobile applications.

Author Contributions: Conceptualization, R.C., F.M. (Francesco Mercaldo) and A.S.; methodology, R.C., F.M. (Francesco Mercaldo), A.S.; software, R.C., F.M. (Francesco Mercaldo); validation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; formal analysis, R.C., F.M. (Francesco Mercaldo) and A.S.; investigation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; resources, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and R. C.; data curation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; writing–original draft preparation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; writing–review and editing, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; visualization, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; supervision, F.M. (Fabio Martinelli) and A.S.; project administration, F.M. (Fabio Martinelli) and A.S.; funding acquisition, F.M. (Fabio Martinelli). All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially supported by MIUR—SecureOpenNets and EU SPARTA and CyberSANE projects.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nguyen, T.; Mcdonald, J.; Glisson, W.; Andel, T. Detecting Repackaged Android Applications Using Perceptual Hashing. In Proceedings of the 53rd Hawaii International Conference on System Sciences, Manoa, HI, USA, 7–10 January 2020.
2. Canfora, G.; Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Leila: Formal tool for identifying mobile malicious behaviour. *IEEE Trans. Softw. Eng.* **2018**, *45*, 1230–1252. [CrossRef]
3. Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Ransomware steals your phone. formal methods rescue it. In *Formal Techniques for Distributed Objects, Components, and Systems*; Springer: Cham, Switzerland, 2016; pp. 212–221.

4. Mercaldo, F.; Visaggio, C.A.; Canfora, G.; Cimitile, A. Mobile malware detection in the real world. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), Austin, TX, USA, 14–22 May 2016; pp. 744–746.
5. Enck, W. Defending users against smartphone apps: Techniques and future directions. In *Information Systems Security*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 49–70.
6. Maiorca, D.; Mercaldo, F.; Giacinto, G.; Visaggio, C.A.; Martinelli, F. R-PackDroid: API package-based characterization and detection of mobile ransomware. In Proceedings of the Symposium on Applied Computing, Marrakech, Morocco, 4–6 April 2017; pp. 1718–1723.
7. Scalas, M.; Maiorca, D.; Mercaldo, F.; Visaggio, C.A.; Martinelli, F.; Giacinto, G. On the effectiveness of system API-related information for Android ransomware detection. *Comput. Secur.* **2019**, *86*, 168–182. [CrossRef]
8. Memon, A.M.; Anwar, A. Colluding apps: Tomorrow's mobile malware threat. *IEEE Secur. Priv.* **2015**, *13*, 77–81. [CrossRef]
9. Marforio, C.; Ritzdorf, H.; Francillon, A.; Capkun, S. Analysis of the communication between colluding applications on modern smartphones. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; pp. 51–60.
10. Casolare, R.; Martinelli, F.; Mercaldo, F.; Santone, A. A Model Checking based Proposal for Mobile Colluding Attack Detection. In Proceedings of the 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 9–12 December 2019; pp. 5998–6000.
11. Xu, K.; Li, Y.; Deng, R.H. Iccdetector: Icc-based malware detection on android. *IEEE Trans. Inf. Foren. Secur.* **2016**, *11*, 1252–1264. [CrossRef]
12. Cimino, M.G.; De Francesco, N.; Mercaldo, F.; Santone, A.; Vaglini, G. Model checking for malicious family detection and phylogenetic analysis in mobile environment. *Comput. Secur.* **2020**, *90*, 101691. [CrossRef]
13. Casolare, R.; Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A. Colluding Android Apps Detection via Model Checking. In Proceedings of the Web, Artificial Intelligence and Network Applications—Proceedings of the Workshops of the 34th International Conference on Advanced Information Networking and Applications, AINA Workshops 2020, Caserta, Italy, 15–17 April 2020; pp. 776–786.
14. Cimitile, A.; Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Talos: No more ransomware victims with formal methods. *Int. J. Inf. Secur.* **2018**, *17*, 719–738. [CrossRef]
15. Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Hey malware, i can find you! In Proceedings of the 2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Paris, France, 13–15 June 2016; pp. 261–262.
16. Blasco, J.; Chen, T.M. Automated generation of colluding apps for experimental research. *J. Comput. Virol. Hacking Tech.* **2018**, *14*, 127–138. [CrossRef]
17. Canfora, G.; De Lorenzo, A.; Medvet, E.; Mercaldo, F.; Visaggio, C.A. Effectiveness of opcode ngrams for detection of multi family android malware. In Proceedings of the 2015 10th International Conference on Availability, Reliability and Security, Toulouse, France , 24–27 August 2015; pp. 333–340.
18. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C. *Drebin: Effective and Explainable Detection of Android Malware in Your Pocket*; NDSS: New York, NY, USA, 2014; Volume 14, pp. 23–26.
19. Li, L.; Bartel, A.; Bissyandé, T.F.; Klein, J.; Le Traon, Y.; Arzt, S.; Rasthofer, S.; Bodden, E.; Octeau, D.; McDaniel, P. Iccta: Detecting inter-component privacy leaks in android apps. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 280–291.
20. Wei, F.; Roy, S.; Ou, X. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur. (TOPS)* **2018**, *21*, 1–32. [CrossRef]
21. Wang, W.; Tian, D.; Meng, W.; Jia, X.; Zhao, R.; Ma, R. MSYM: A multichannel communication system for android devices. *Comput. Netw.* **2020**, *168*, 107024. [CrossRef]
22. Al-Haiqi, A.; Ismail, M.; Nordin, R. A new sensors-based covert channel on android. *Sci. World J.* **2014**, *2014*, 969628. [CrossRef] [PubMed]
23. Bugiel, S.; Davi, L.; Dmitrienko, A.; Fischer, T.; Sadeghi, A.R. *Xmandroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*; Technical Report TR-2011-04; Technische Universität Darmstadt: Darmstadt, Germany, 2011.

24. Tiwari, A.; Groß, S.; Hammer, C. IIFA: Modular inter-app intent information flow analysis of android applications. In *Security and Privacy in Communication Systems*; Springer: Cham, Switzerland, 2019; pp. 335–349.
25. Bosu, A.; Liu, F.; Yao, D.; Wang, G. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, UAE, 2–6 April 2017; pp. 71–85.
26. Spolaor, R.; Abudahi, L.; Moonsamy, V.; Conti, M.; Poovendran, R. No free charge theorem: A covert channel via usb charging cable on mobile devices. In *Applied Cryptography and Network Security*; Springer: Cham, Switzerland, 2017; pp. 83–102.
27. Liu, F.; Cai, H.; Wang, G.; Yao, D.; Elish, K.O.; Ryder, B.G. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW), San Jose, CA, USA, 25 May 2017; pp. 189–198.
28. Enck, W.; Gilbert, P.; Han, S.; Tendulkar, V.; Chun, B.G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst. (TOCS)* **2014**, *32*, 1–29. [CrossRef]
29. Asavoae, I.M.; Blasco, J.; Chen, T.M.; Kalutarage, H.K.; Muttik, I.; Nguyen, H.N.; Roggenbach, M.; Shaikh, S.A. Towards automated android app collusion detection. *arXiv* **2016**, arXiv:1603.02308.
30. Wong, M.Y.; Lie, D. *IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware*; NDSS: New York, NY, USA, 2016; Volume 16, pp. 21–24.
31. Hay, R.; Tripp, O.; Pistoia, M. Dynamic detection of inter-application communication vulnerabilities in Android. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 12–17 July 2015; pp. 118–128.
32. Bhandari, S.; Jaballah, W.B.; Jain, V.; Laxmi, V.; Zemmari, A.; Gaur, M.S.; Mosbah, M.; Conti, M. Android inter-app communication threats and detection techniques. *Comput. Secur.* **2017**, *70*, 392–421. [CrossRef]