applied sciences

MDPI

Article

Detecting Colluding Inter-App Communication in Mobile Environment

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

1 Department of Biosciences and Territory, University of Molise, 86090 Pesche, Italy
2 Institute for Informatics and Telematics, National Research Council of Italy, 56124 Pisa, Italy; fabio.martinelli@iit.cnr.it
3 Department of Medicine and Health Sciences "Vincenzo Tiberio", University of Molise, 86100 Campobasso, Italy; antonella.santone@unimol.it
* Correspondence: rosangela.casolare@unimol.it (R.C.); francesco.mercaldo@unimol.it or francesco.mercaldo@iit.cnr.it (F.M.)
† All authors contributed equally to this work.

Received: 30 September 2020; Accepted: 16 November 2020; Published: 24 November 2020

Abstract: The increase in computing capabilities of mobile devices has, in the last few years, made possible a plethora of complex operations performed from smartphones and tablets end users, for instance, from a bank transfer to the full management of home automation. Clearly, in this context, the detection of malicious applications is a critical and challenging task, especially considering that the user is often totally unaware of the behavior of the applications installed on their device. In this paper, we propose a method to detect inter-app communication i.e., a colluding communication between different applications with data support to silently exfiltrate sensitive and private information. We based the proposed method on model checking, by representing Android applications in terms of automata and by proposing a set of logic properties to reduce the number of comparisons and a set of logic properties automatically generated for detecting colluding applications. We evaluated the proposed method on a set of 1092 Android applications, including different colluding attacks, by obtaining an accuracy of 1, showing the effectiveness of the proposed method.

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

1. Introduction

The growing technological development of recent years and the evolution of communication devices have changed our daily habits. The introduction of smartphones and tablets in our lives has allowed greater globalization, in fact, anyone can access the same information at the same time, thus reducing cultural differences and barriers.

However, the technology spread too fast due to the possibility of buying models of smartphones (or tablets) of medium–high quality at affordable prices but also allowing users to download many applications freely.

As result of this rapid spread, users have not had adequate "training" for the correct use of devices. People use smartphones without thinking about the problems related to their use, because these devices handle data of different types everyday, among which, we have personal information, the principal target of cybercriminals [1,2].

Among the various operating systems for the mobile environment, the most popular is Android. Android is widespread and has an "open nature", allowing attackers to hit a significant amount of users quite easily [3,4].

On mobile devices powered by the Android operating system, it is possible to download applications of various kinds and with different functionalities. Usually, people consider the official marketplace (for instance, Google Play in the Android environment) when attempting to download desired applications, but sometimes, they use unofficial markets (for instance, AppChina) to obtain applications for free, which usually require payment on the official store, or users will install applications not available on the official Android market [5].

In most cases, third-party markets turn out to be untrustworthy [6]; however, we can find infected applications even in official markets [7]. In this scenario, there are two main actors: the *attackers*, characterized by cybercriminals who implement malicious code to attack users and steal their information, and the *defenders*, characterized instead by creating tools, usually called anti-malware, that are able to detect the presence of threats [8]. Anti-malware applications, however, are unable to identify new threats, since recognition can only take place if the information about the threat is already known and stored in their repository [9].

In order to increase the complexity of the malicious payloads (i.e., the code aimed to perform the harmful action) and thus be capable of evading anti-malware programs, the malware writers developed a new threat called *Colluding Attack*. A colluding attack consists of dividing the malicious action between two or more applications. In this way, the anti-malware, analyzing any single application will not able to identify the potential threat, since it is the communication between these applications that will launch the attack.

The attack occurs in the following way: we have two infected applications that communicate with each other when the user performs a certain action or when an event occurs in the system. The first application reads the user's sensitive data and then sends them to the second one, which then transmits them to a 3rd party. We can deduce that the first application has the permission to read the data, while the second one has the permission to connect to a network, thus sharing the information.

In the Android environment, the applications may communicate through the Inter-Component Communication (ICC): this mechanism is useful for the developer by allowing functionality reuse [10]. The presence of this mechanism means that applications are not always independent from each other, because the inter-application collaboration expects an information exchange between components belonging to the same application or to different applications [11]. The second possibility mentioned can be used by malicious applications to attack users' data [12,13].

To detect and verify the presence of colluding applications on Android, we developed a method based on the use of model checking techniques [9]. We propose an heuristic function that reduces the number of candidate applications to the analysis. The function is defined based on the μ-calculus temporal logic and model checking. In this work, we focus our attention on different ICC, in particular: *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, and *Remote Procedure Calls* (RPC).

This kind of communications can be used to launch an attack, as follows:

- with the *SharedPreferences*, it is possible to store key–value pairs of data, configuration, and preferences. So an application can save some settings (containing data information) into a shared preference file, which could be read by the receiving application;
- with the *ExternalStorage*, we consider the ability to store the information in a file. In this case, for example, an application can read data and send them to a second application via an intent. The second application, using the external storage, can send the data though an external network (for instance, Internet);
- with the *BroadcastReceiver*, it is possible to be notified of an event occurrence at the system level (e.g., receiving an SMS). It is useful for the instant management of special events. Broadcast receivers respond to broadcast messages sent using Intent objects, which can come from the same application or from others applications installed on the device. So, with this component, it is possible to have applications that receive data both via broadcast receivers and SMS messages [10];
- with the *RPC*, it is possible for an Android application to call a method to execute in a different address space (typically on another Android application), which is coded as if it were a normal

(i.e., local) procedure call, without the developer explicitly coding the implementations for the remote interaction: the developer writes the same code whether the subroutine is local to the executing application or remote. Android provides several distinctive components, for instance, Activity and Service, which are able to perform this kind of communication by also sharing data between these components. In this attack, the collusion happens when an Android component (for instance an Activity) starts a component in another application (by invoking a Service in the second application) by also sharing data between the Activity and the Service components.

We chose ICC attacks since they are the most common ones used to launch colluding interactions, as demonstrated in [14–17].

The detection of colluding behavior in the Android environment, by exploiting model checking, was already explored by the authors of [18]. Below, we itemize the main points introduced in this paper with respect to the work in [18]:

- in this paper, we consider the detection of four colluding attacks (i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, and *RPC*) whereas the work in [18] focused only on the *SharedPreferences* colluding attack. This represents the main contribution of the work and, at the best of knowledge authors, this is the first contribution devoted to the detection of these four different harmful colluding attacks;
- we propose an algorithm for the automatic detection of collusion. The authors in [18] propose a set of manually designed properties for the detection of *SharedPreferences* collusion. In this work, we propose an algorithm for the automatic generation of properties for the detection of whether two or more applications are performing a colluding attack through *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, or *RPC*;
- the proposed method can also detect colluding attacks performed by more than two applications, whereas the method in [18] can detect collusion between only two applications. This is important, as colluding attacks are usually perpetrated by more than two applications [16,19] in order to have more chances to activate the malicious behavior;
- we developed and evaluated 20 (10 aimed to send data and the remaining 10 for data receiving) Android applications performing a colluding attack by exploiting the *RPC* communication channel;
- we experimented with an extended set of colluding and not-colluding Android applications to evaluate the proposed method, also taking into account the colluding applications aimed to share information by exploiting *ExternalStorage*, *BroadcastReceiver*, and *RPC*, and we obtain better performance-related results as compared to [18], where an accuracy equal to 0.99 is obtained; our proposed method reaches an accuracy equal to 1.

It should be noted that the other works in this research field are focused on the analysis of one application at a time, whereas our method, by analyzing multiple applications together, allows us to identify the presence of an inter-application communication. Furthermore, the other research base their work on the use of machine learning techniques, unlike ours, which is based on model checking techniques. The paper proceeds as follows: in the next section, we provide background information regarding the model checking technique, so that readers can fully grasp the information presented in this paper; in Section 3, the proposed method for the detection of colluding inter-app communication is presented; Section 4 presents the experimental analysis results to demonstrate the effectiveness of the proposed approach; the current literature regarding collusion attack detection is discussed in Section 5; in the last section, conclusions and future research directions are presented.

2. Model Checking Background

Preliminary concepts about the model checking technique implemented in our proposed approach are provided below. To apply model checking, we first represent a system using a formal specification language. In the following, we use the calculus of communicating systems (CCS) in a slightly extended

form [20]. Systems are described in terms of processes. Processes perform actions, evolving to become new (and usually different) processes after each action. The syntax to describe processes is shown in (1):

$$p ::= DONE \mid x \mid \alpha.p \mid p + p \mid p; p \mid p \| p \mid p \backslash L \mid p[f] \tag{1}$$

where α ranges over a finite set of actions $\mathcal{A} = \{\tau, a, \overline{a}, b, \overline{b}, ...\}$. The action $\tau \in \mathcal{A}$ is called the *internal action*. The set of *visible actions*, $\mathcal{V}$, ranged over by l, is defined as $\mathcal{A} - \{\tau\}$. The set L, in processes of the form $p \backslash L$, is a set of actions such that $L \subseteq \mathcal{V}$; whereas the relabeling function f, in processes of the form $p[f]$, is a total function, $f : \mathcal{A} \to \mathcal{A}$, such that the constraint $f(\tau) = \tau$ is respected. Each action $l \in \mathcal{V}$ (resp. $\overline{l} \in \mathcal{V}$) has a *complementary action* $\overline{l}$ (resp. l). It holds that $\overline{\overline{l}} = l$. Given $L \subseteq \mathcal{V}$, with L^+ we denote the set $\{\overline{l}, l \mid l \in L\}$. Variable x ranges over a set of *constant* names: each constant x is defined by a constant definition $x \stackrel{\text{def}}{=} p$, where p is called the *body* of x. $DONE$ is the constant defined as $\delta.nil$ that corresponds to a process whose task is to terminate performing only the action δ. The process *nil* cannot perform any action; it is also said to be "deadlocked". We denote the set of all processes by $\mathcal{P}$.

Given a set $\mathcal{D}$ of constant definitions, the standard *operational semantics* $\mathcal{S}$ is given by a relation $\longrightarrow_{\mathcal{D}} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$. $\longrightarrow_{\mathcal{D}}$ ($\longrightarrow$ for short) is the least relation defined by the rules in Table 1. Given a process p the semantic of p is the automaton is called *standard transition system* for p and is denoted $\mathcal{S}(p)$.

Table 1. Operational semantics for the extended calculus of communicating systems (CCS).

Done	$\dfrac{}{DONE \xrightarrow{\delta} nil}$ (2)	**Act**	$\dfrac{}{\alpha.p \xrightarrow{\alpha} p}$ (3)
Seq$_1$	$\dfrac{p \xrightarrow{\alpha} p'}{p; q \xrightarrow{\alpha} p'; q}\ \alpha \neq \delta$ (4)	**Seq**$_2$	$\dfrac{p \xrightarrow{\delta} nil}{p; q \xrightarrow{\delta} q}$ (5)
Sum	$\dfrac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'}$ (and symmetric) (6)	**Par**	$\dfrac{p \xrightarrow{\alpha} p'}{p \| q \xrightarrow{\alpha} p' \| q}\ \alpha \neq \delta$ (and symmetric) (7)
Com$_2$	$\dfrac{p \xrightarrow{\delta} p',\ q \xrightarrow{\delta} q'}{p \| q \xrightarrow{\delta} DONE}$ (8)	**Com**$_1$	$\dfrac{p \xrightarrow{l} p',\ q \xrightarrow{\overline{l}} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$ (9)
Res	$\dfrac{p \xrightarrow{\alpha} p'}{p \backslash L \xrightarrow{\alpha} p' \backslash L}\ \alpha \notin L^+$ (10)	**Rel**	$\dfrac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$ (11)
Con	$\dfrac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \stackrel{\text{def}}{=} p$ (12)		

We now informally explain the semantics of an extended CCS process. Note that there is no rule for the process *nil*, which thus cannot perform any action. In the **Done** rule, shown in (2) in Table 1, the process can perform δ and then reaches a deadlocked state.

In **Act** rule, shown in (3) in Table 1, the process $\alpha.p$ can perform the action α to become the process p.

Seq$_1$ and **Seq**$_2$ rules represent the sequentialization of two processes, as shown in (4) and (5) in Table 1. The q process can start its execution only when the p process has terminated its execution by performing the δ action.

The **Sum** rule, shown in (6) in Table 1, states that p and q are alternative choices for the behavior of $p + q$. The operator $\|$ expresses the parallel execution. The **Par** rule, shown in (7) in Table 1, shows how processes in a parallel composition can behave autonomously: if the process p performs α and becomes p', then $p \| q$ performs α and becomes $p' \| q$ (similarly for q). When **Com**$_1$ and **Com**$_2$ rules

(shown in (8) and (9) in Table 1) is used, we say that a *handshake* occurs. A handshake occurs only if two processes can simultaneously execute the complementary actions; the handshake results in an internal communication (the action τ). When both possesses p and q have terminated their execution, the $p \| q$ process becomes $DONE$. The operator $\backslash L$, in the **Res** rule (shown in (9) in Table 1), prevents actions in L^+ to be done: if p can perform α to become p', then $p \backslash L$ can perform α to become $p' \backslash L$ only if $\alpha \notin L^+$. In the **Rel** rule, the operator $[f]$ renames actions by means of the relabeling function f: if p can perform α to become p', then $p[f]$ can perform $f(\alpha)$ to become $p'[f]$. Finally, a constant x behaves as p if $x \stackrel{\text{def}}{=} p$ as stated in the rule **Con**, (shown in (12) in Table 1). Roughly speaking, the **Con** rule states that a process behaves like its definition.

Once we obtain the formal model of a system S, we have to prove properties about S. This is accomplished by using a temporal logic [21]. We use *mu-calculus* logic [21]—the syntax is reported in (13). We suppose that Z ranges over a set of variables, K and R range over sets of actions $\mathcal{A}$.

$$\begin{aligned} \phi \quad ::= \quad & \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \vee \phi \mid \phi \wedge \phi \mid \\ & [K]\,\phi \mid \langle K \rangle\,\phi \mid \nu Z.\phi \mid \mu Z.\phi \end{aligned} \quad (13)$$

The satisfaction of a formula ϕ by a state s of a transition system, denoted by $s \models \phi$, is so defined:

- each state satisfies `tt` and no state satisfies `ff` (as shown by (14) in Table 2);
- a state satisfies $\phi_1 \vee \phi_2$ ($\phi_1 \wedge \phi_2$) if it satisfies ϕ_1 or (and) ϕ_2 (as shown by (15) in Table 2).
- $[K]\,\phi$ and $\langle K \rangle\,\phi$ are the modal operators (as shown by (17) and (18) in Table 2): $[K]\,\phi$ is satisfied by a state, which, for every performance of an action in K, evolves in a state obeying ϕ; while $\langle K \rangle\,\phi$ is satisfied by a state, which can evolve to a state obeying ϕ by performing an action in K.

In Table 2 is reported the precise definition of the satisfaction of a closed formula φ by a state s (denoted $s \models \varphi$).

Table 2. Satisfaction of a closed formula by a state.

$p \not\models \mathtt{ff}$	and	$p \models \mathtt{tt}$	(14)
$p \models \varphi \wedge \psi$	iff	$p \models \varphi$ and $p \models \psi$	(15)
$p \models \varphi \vee \psi$	iff	$p \models \varphi$ or $p \models \psi$	(16)
$p \models [K]\,\varphi$	iff	$\forall p'.\forall \alpha \in K.p \xrightarrow{\alpha} p'$ implies $p' \models \varphi$	(17)
$p \models \langle K \rangle\,\varphi$	iff	$\exists p'.\exists \alpha \in K.p \xrightarrow{\alpha} p'$ and $p' \models \varphi$	(18)
$p \models \nu Z.\varphi$	iff	$p \models \nu Z^n.\varphi$ for all n	(19)
$p \models \mu Z.\varphi$	iff	$p \models \mu Z^n.\varphi$ for some n	(20)

where:

- for each n, $\nu Z^n.\varphi$ and $\mu Z^n.\varphi$ are defined as:

$$\begin{aligned} \nu Z^0.\varphi &= \mathtt{tt} & \mu Z^0.\varphi &= \mathtt{ff} \\ \nu Z^{n+1}.\varphi &= \varphi[\nu Z^n.\varphi / Z] & \mu Z^{n+1}.\varphi &= \varphi[\mu Z^n.\varphi / Z] \end{aligned}$$

where the notation $\varphi[\psi/Z]$ indicates the substitution of ψ for every free occurrence of the variable Z in φ.

$\mu Z.\phi$ and $\nu Z.\phi$ are the fixed point formulae, where μZ (νZ) (as shown by (19) and (20) in Table 2) *binds* free occurrences of Z in ϕ. An occurrence of Z is free if it is not within the scope of a binder μZ (νZ). A formula is *closed* if it contains no free variables. $\mu Z.\phi$ is the least fix-point of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one. A transition system T satisfies a formula ϕ, denoted $T \models \phi$, if and only if $q \models \phi$, where q is the initial state of T. A CCS process p satisfies ϕ if $S(p) \models \phi$.

In (21)–(23), we consider the following abbreviations (where K ranges over sets of actions and $\mathcal{A}$ is the set of all actions):

$$\langle \alpha_1, \ldots, \alpha_n \rangle \varphi \stackrel{\text{def}}{=} \langle \{\alpha_1, \ldots, \alpha_n\} \rangle \varphi \quad (21)$$

$$\langle - \rangle \varphi \stackrel{\text{def}}{=} \langle \mathcal{A} \rangle \varphi \quad (22)$$

$$\langle -K \rangle \varphi \stackrel{\text{def}}{=} \langle \mathcal{A} - K \rangle \varphi \quad (23)$$

Finally, once defined the model and the temporal logic properties, we need something enabling us to check whether the model satisfies the defined properties. To this aim, formal verification is considered, a system process exploiting mathematical reasoning to verify if a system (i.e., the model) satisfies some requirement (i.e., the temporal logic properties).

In recent years, several verification techniques were proposed, e.g., in [22], model checking is considered.

In the model checking technique, the properties are formulated in temporal logic: each property is evaluated against the system i.e., the Labeled Transition System (LTS)-based model. The model checker accepts as input a model and a property, it returns "true" whether the system satisfies the formula and "false" otherwise. The performed check is an exhaustive state space search that is guaranteed to terminate since the model is finite. In this paper, we use the Concurrency WorkBench of the New Century (CWB-NC) (https://www3.cs.stonybrook.edu/~cwb/), a widespread formal verification environment. The (CWB-NC) is our model checker and checks the presence of the PUT and GET properties in the applications, which finally verifies if there is a collusion. Note that we can easily use CWB-NC for the extended CCS since, as shown in [20], the new operators can be easily translated using the standard CCS ones.

3. Colluding Inter-App Communication Detection

In this section, we present the proposed method for the detection of Android-based colluding applications. We remark that our proposal represents the first method designed for the detection of different kinds of colluding attacks: *ExternalStorage*, *SharedPreferences*, *BroadcastReceiver*, and *RPC*.

Figure 1 shows an overview about the proposed approach.

We began our investigation with an Android *Device*, where the installed applications are gathered by obtaining the APKs at the */data/app* url in the device internal memory that is possible to read without root permission [23]: the permissions on that directory are *rwxrwx–x*. The APK extension indicates an Android Package file. This file format, basically a variant of the JAR file, is considered for the distribution and installation of bundled components on the Android mobile platform [24]. The APK contains all the resources for the application running, for instance, external libraries, images, audio, and the set of .class files of the application, stored in the *classes.dex* file, in a format translated for Dalvik, the Android virtual machine.

Once we obtain the APKs of the *Installed Applications*, through a reverse engineering process, we obtain the Java bytecode for each APK. We obtain the Java bytecode representation for each method of Android applications by invoking the *dex2jar* (https://github.com/pxb1988/dex2jar) tool to obtain from the classes.dex file (the executable file targeting the Android virtual machine) the .jar one (the executable targeting the Java virtual machine), and the Byte Code Engineering Library (BCEL) (https://commons.apache.org/proper/commons-bcel/), to obtain the classes stored in the .jar file their Java bytecode representations. We consider the Java bytecode because is already possible to obtain even the APKs were obfuscated [25–28], for instance, using the R8 compiler built-in in the last Android studio release (https://developer.android.com/studio/build/shrink-code), providing a kind of obfuscation to shorten the names of the classes and methods, but also optimization techniques that can make the code more difficult to read, applying aggressive strategies to reduce the size of the developed application.

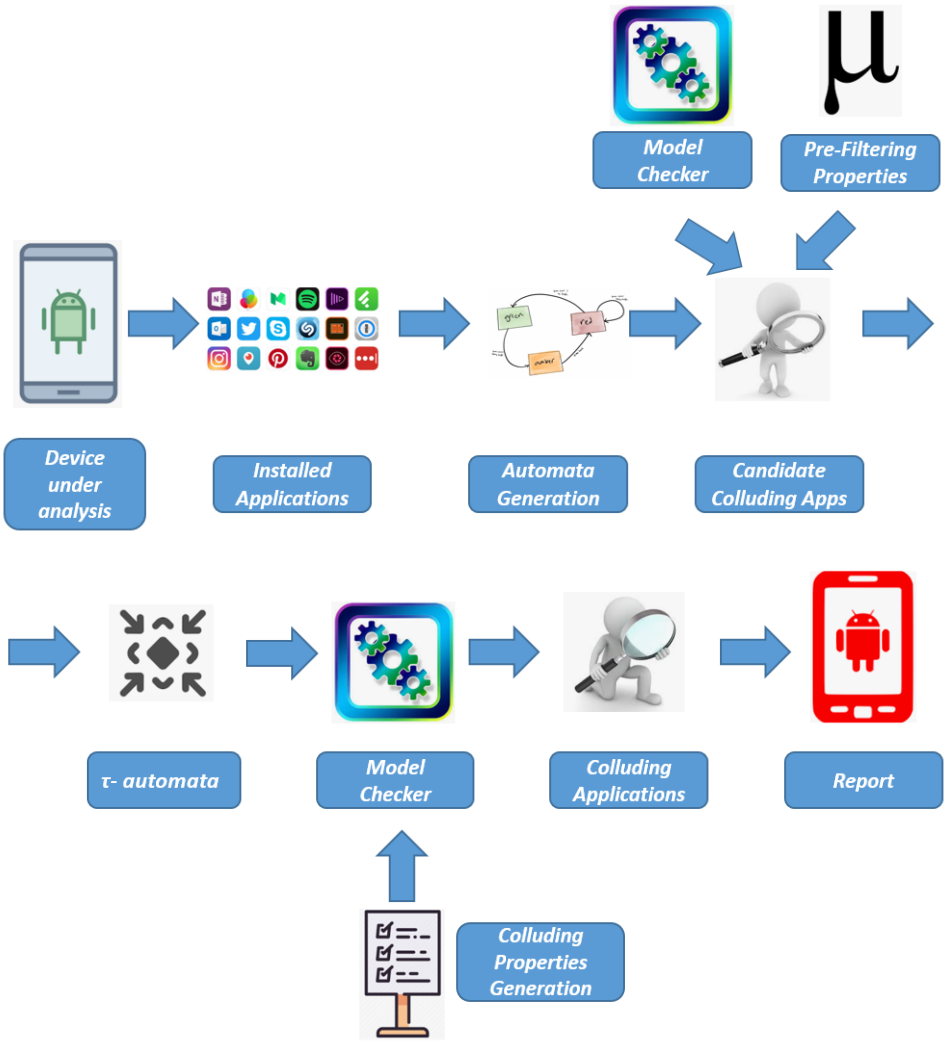

Figure 1. The proposed approach for colluding application detection.

Once we obtained the Java bytecode related to the installed APKs, we generated a model for each application (i.e., *Automata Generation* step in Figure 1). In the following, we explain how we generate the automata. We considered a CCS model for each method of the APK. This is obtained by translating each Java bytecode instruction in a CCS process. The precise definition of the translation can be found in [6,7].

With regard to the sequential Java bytecode instructions, the translation is the following:

$$proc\ x_{current} = opcode.x_{next}$$

where $x_{current}$ represents the current instruction under analysis, whereas x_{next} is the process representing the successive instruction and finally, $opcode$ is the name of the Java bytecode instruction. Note that to express constant definitions we use the syntax of the Concurrency Workbench of New Century [20], the formal verification environment used for the experimentation. Thus, instead of $x \stackrel{\text{def}}{=} p$ we write $proc\ x\ =\ p$.

An example of CCS translation from the translation of sequential op-code instructions is shown in Listings 1 and 2.

Listing 1. CCS process for Listing 1.

```
proc P1 = store.M2
proc P2 = load.M3
proc P3 = return.nil
```

Listing 2. CCS process for Listing 2.

```
proc P1 = dup.M2
proc P2 = invokesubstring.M3
proc P3 = pushConstant.M4
proc P4 = newStringBuilder.M5
proc P5 = invokeinit.M6
proc P6 = pop.M7
proc P7 = return.nil
```

In Listing 1, there are only three actions i.e., store, load, and return, whereas in Listing 2, there are more actions, for instance, related to method invocations (i.e., invokesubstring and invokeinit) to the definition of new Java objects (i.e., newStringBuilder), stack instruction as pop (to discard the top value on the stack) or dup (to duplicate the value on top of the stack), but also the push instruction, aimed to push a variable into the stack (in this case the variable is "Constant").

Branch instructions are used to change the sequence of the instruction execution. We consider, as explained in Section 2, the $+$ operator to manage the choice.

A CCS process was built for each method of the application under analysis. Let *aua* be an application under analysis. Supposing that the *aua* has n methods, i.e., $F_1, \ldots, F_n$, the *aua* CCS representation has n $M_1, \ldots, M_n$ CCS processes.

Once generated, the automata, in terms of CCS process, are considered as an input to the Model Checker with a set of formulae aimed to verify whether the applications exhibit a GET or a PUT operation related to an *ExternalStorage*, *SharedPreferences*, *BroadcastReceiver*, or *RPC*. For this verification, we exploited a series of *Pre Filtering Properties*. The automata that give the result TRUE from the Model Checker will compose the *Candidate Colluding Apps*.

With regard to the *SharedPreferences*, and similarly for the *ExternalStorage*, the *BroadcastReceiver*, and the *RPC*, an application can execute two different operations on a shared resource: PUT and GET. For the *SharedPreferences*, we encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a shared resource, the formula (Table 3—*Formula_SP_PUT*) results true if the following actions are performed: *invokegetSharedPreferences*, *invokeedit*, *invokeputString*/*invokeputInt*/*invokeputFloat*, and *invokecommit*;
- when instead an application executes a GET action on a shared resource, the formula (Table 3—*Formula_SP_GET*) results true if the following actions are performed: *invokegetSharedPreferences*, *invokegetString*/*invokegetInt*/*invokegetFloat*.

Table 4 shows the χ_{PUT} and the χ_{GET} detect the PUT and GET operations, respectively, on the *ExternalStorage*. In this case, we encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a shared resource, the formula (Table 4—*Formula_ES_PUT*) results true if the following actions are performed: *invokegetExternalStorageDirectory* and *invokewrite*;
- when instead an application executes a GET action on a shared resource, the formula (Table 4—*Formula_ES_GET*) results true if the following actions are performed: *invokegetExternalStorageDirectory* and *invokereadFully*.

Table 3. The φ_{PUT} property detects methods invoking *PUT* operations on *SharedPreferences*, while the φ_{GET} property detects methods invoking *GET* operations on *SharedPreferences*.

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

Table 4. The χ_{PUT} property detects methods invoking *PUT* operations on *ExternalStorage*, while the χ_{GET} property detects methods invoking *GET* operations on *ExternalStorage*.

Formula_ES_PUT
$\chi_{PUT} = \mu X.\langle invokegetExternalStorageDirectory\rangle\ \chi_{PUT_1} \vee$
$\langle -invokegetExternalStorageDirectory\rangle\ X$
$\chi_{PUT_1} = \mu X.\langle invokewrite\rangle\ \mathtt{tt} \vee$
$\langle -invokewrite\rangle\ X$
Formula_ES_GET
$\chi_{GET} = \mu X.\langle invokegetExternalStorageDirectory\rangle\ \chi_{GET_1} \vee$
$\langle -invokegetExternalStorageDirectory\rangle\ X$
$\chi_{GET_1} = \mu X.\langle invokereadFully\rangle\ \mathtt{tt} \vee$
$\langle -invokereadFully\rangle\ X$

Table 5 shows the ψ_{PUT} and the ψ_{GET} detects PUT and GET operations, respectively, on the *BroadcastReceiver*. In this case, we encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a shared resource, the formula (Table 5—*Formula_BR_PUT*) results true if the following actions are performed: *invokeputExtra*, *invokesendBroadcast*;
- when instead an application executes a GET action on a shared resource, the formula (Table 5—*Formula_BR_GET*) results true if the following actions are performed: *invokegetStringExtra*, *invokestart*.

Table 5. The ψ_{PUT} property detects methods invoking *PUT* operations on *BroadcastReceiver*, while the ψ_{GET} property detects methods invoking *GET* operations on *BroadcastReceiver*.

Formula_BR_PUT
$\psi_{PUT} = \mu X.\langle invokeputExtra\rangle\ \psi_{PUT_1} \vee$
$\langle -invokeputExtra\rangle\ X$
$\psi_{PUT_1} = \mu X.\langle invokesendBroadcast\rangle\ \mathtt{tt} \vee$
$\langle -invokesendBroadcast\rangle\ X$
Formula_BR_GET
$\psi_{GET} = \mu X.\langle invokegetStringExtra\rangle\ \psi_{GET_1} \vee$
$\langle -invokegetStringExtra\rangle\ X$
$\psi_{GET_1} = \mu X.\langle invokestart\rangle\ \mathtt{tt} \vee$
$\langle -invokestart\rangle\ X$

In Table 6, we show the ρ_{PUT} and the ρ_{GET} detect PUT and GET operations, respectively, when the *RPC* channel is exploited. In this case, we encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a Remote Procedure Call (RPC) channel, the formula (Table 6—*Formula_RPC_PUT*) results true if the following actions are performed: *invokeputExtra*, *invokestartService*;
- when instead an application executes a GET action on a RPC channel, the formula (Table 6—*Formula_RPC_GET*) results true if the following actions are performed: *invokegetStringExtra* and *invokevirtual*.

Table 6. The ρ_{PUT} property detects methods invoking *PUT* operations on *RemoteProcedureCalls*(*RPCs*), while the ρ_{GET} property detects methods invoking *GET* operations by exploiting *RPC*.

Formula_RPC_PUT
$\rho_{PUT} = \mu X.\langle invokeputExtra\rangle\ \rho_{PUT_1} \vee$
$\langle -invokeputExtra\rangle\ X$
$\rho_{PUT_1} = \mu X.\langle invokestartService\rangle\ \mathtt{tt} \vee$
$\langle -invokestartService\rangle\ X$
Formula_RPC_GET
$\rho_{GET} = \mu X.\langle invokegetStringExtra\rangle\ \rho_{GET_1} \vee$
$\langle -invokegetStringExtra\rangle\ X$
$\rho_{GET_1} = \mu X.\langle invokevirtual\rangle\ \mathtt{tt} \vee$
$\langle -invokevirtual\rangle\ X$

We highlight that the CWB-NC is exploited in two different times by the proposed method: the first one to detect the methods candidate for the collusion (by detecting *GET* and *PUT*), and the second time to detect the applications that collude with each other, by identifying also the shared resources.

The idea behind these properties is to obtain in a short time window four different sets of applications, each set related to a different collusion attacks (i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, and *RPC*), and in each set we have the classes (i.e., the CCS automata) verifying the PUT and the *GET* properties for each collusion attack. 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.

Once obtained from the *Installed Applications* the set of the applications that can potentially exhibit colluding behavior, the next steps depicted in Figure 1 detects whether two or more applications can effectively share information, thus performing a collusion attack.

Thus, for each automata belonging to the *Candidate Colluding Apps*, we generate a simplified version containing only the action that can be involved in the collusion attack: for this reason, the actions that cannot be involved in the collusion attacks are set with τ actions. In particular, the models resulting from the τ-automata step contains the push action, symptomatic of read and write operations on a variable and a set of actions discriminating the *SharedPreferences*, the *ExternalStorage*, the *BroadcastRecevier*, and the *RPC* colluding attacks. From the τ-automata models we automatically generate a set of properties, aimed to verify whether two or more applications can effectively perform a colluding attack.

Let us better understand how we built the τ-automata with an example. In Listing 3, we show an example of automaton related to a PUT operation of an *ExternalStorage* collusion attack, and in Listing 4, we show the related τ-automaton.

Listing 3. CCS process for Listing 3

```
proc M1 = invokeinit.M2
proc M2 = invokegetExternalStorage.M3
proc M3 = pushConstant1.M4
proc M4 = store.M5
proc M5 = pushConstant2.M6
proc M6 = load.M7
proc M7 = pushConstant3.M8
proc M8 = invokewrite.M9
proc M9 = return.nil
```

Listing 4. CCS process for Listing 4

```
proc M1 = t.M2
proc M2 = invokegetExternalStorage.M3
proc M3 = pushConstant1.M4
proc M4 = t.M5
proc M5 = pushConstant2.M6
proc M6 = t.M7
proc M7 = pushConstant3.M8
proc M8 = t.M9
proc M9 = t.nil
```

The automaton shown in Listing 3 results in *true* to the property related to the *ExternalStorage* PUT (i.e., χ_{GET} in Table 4): as a matter of fact it shows an *invokegetExternalStorage* action and, after an undefined number of actions, the *invokewrite* one. Listing 4 shows the τ-automaton related to the automaton shown in Listing 3: all the actions are translated into τ actions (not of interest for the collusion detection), with the exception of the actions involving a *push* operation (in this example *pushConstant*1, *pushConstant*2, and *pushConstant*3, respectively, to push into the stack the Constant1, Constant2, and Constant3 variables) and the *invokegetExternalStorage* one. In fact, we recall that for the GET and PUT automata, in addition to the push actions, the τ-automata consider the *invokegetExternalStorage* action for the *ExternalStorage* colluding attack whereas, for the *SharedPreferences* attack we consider the *invokegetSharedPreferences* action and, for the *BroadcastReceiver* attack we consider the *invokeputExtra*, *invokeSendBoradcast* (for the PUT), *invokeStringExtra*, *invokeStart* (for the GET), and the push under analysis (for both the GET and PUT *BroadcastReceiver* properties).

Once obtained, all the τ-automata for the GET and the PUT operations for the three collusion attacks we considered, we designed an algorithm, the flowchart of which is shown in Figure 2 for the automatic *Colluding Properties Generation* step, in particular, for the generation of the properties for the *ExternalStorage*, the *SharedPreferences*, the *BroadcastReceiver*, and the *RemoteProcedureCall* colluding detection.

The idea behind the designed algorithm is to automatically infer a set of properties and, whether a PUT and a GET automata (related to *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, or the *RemoteProcedureCall*) result in true to the properties including the same push action, in this case, we mark the PUT and the GET automata as performing a colluding attack.

Below, we explain the algorithm steps in detail by considering the flowchart shown in Figure 2.

For each attack, we consider i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, and *RPC*, we take as input the set of τ-automata related to the PUT operation (*set of PUT t-automata* in the flowchart in Figure 2). For each PUT automaton, we obtain all the PUSH operations (by performing a *sort* operation for displaying the set of visible actions of the automata (http://courses.cs.vt.edu/

cs5204/fall00/CWB/top-level-coms.html)) and, for each action considering a push operation we automatically generate rules involved the push and following actions:

- with regard to the *SharedPreferences*, we consider the *invokegetSharedPreferences* and the push under analysis;
- with regard to the *ExternalStorage*, we consider the *invokegetExternalStorage* and the push under analysis;
- with regard to the *BroadcastReceiver*, we consider the *invokeputExtra*, *invokeSendBoradcast* (for the PUT property), *invokeStringExtra*, and *invokeStart* (for the GET property);
- with regard to the *RemoteProcedureCall*, we consider the *invokeputExtra*, *invokestartService* (for the PUT property), *invokegetStringExtra*, and *invokevirtual* (for the GET property).

The properties are automatically generated by looking only the PUT τ-automata, as shown from the flowchart in Figure 2.

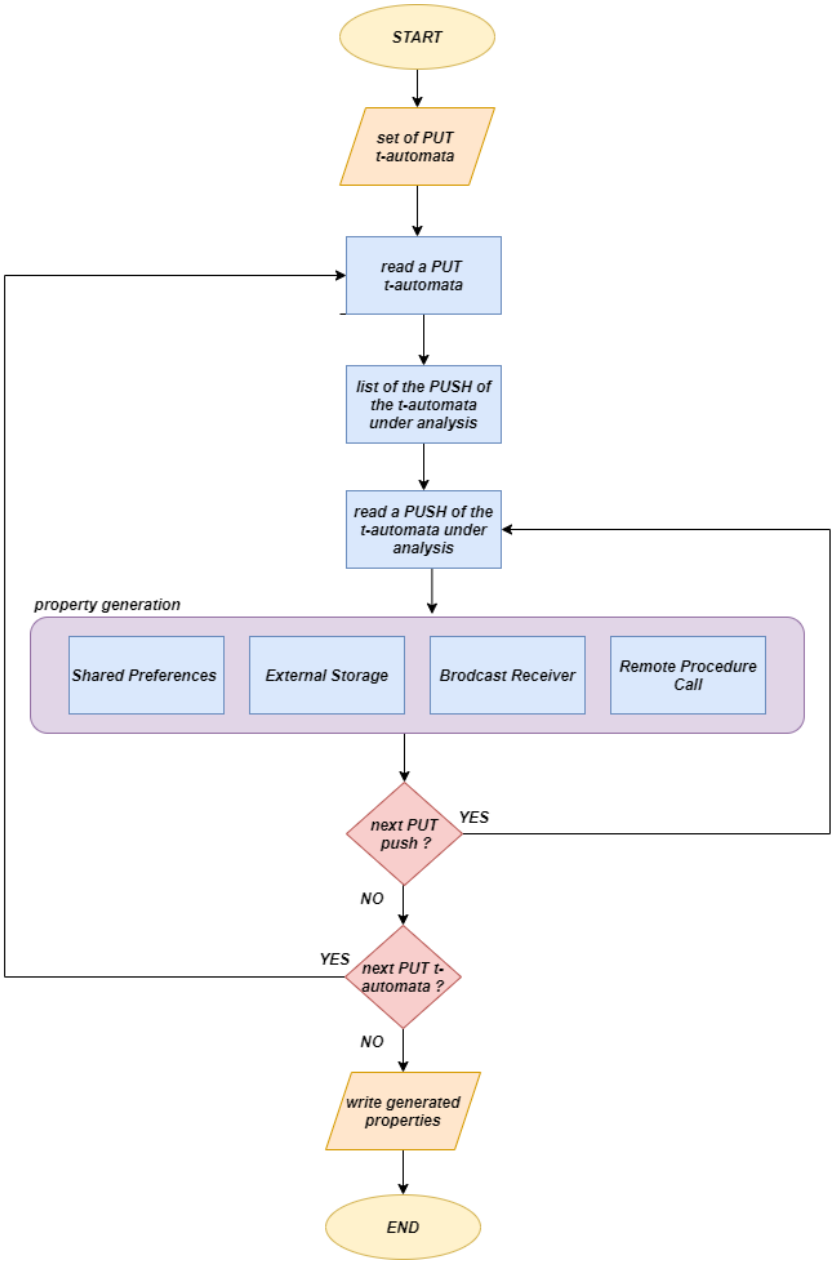

Figure 2. The colluding properties generation step.

For instance, considering the PUT *ExternalStorage* τ-automaton shown in Listing 4, the properties automatically generated by the proposed algorithm are shown in Table 7: for each push action of the τ-automaton a property is generated.

Table 7. The properties automatically generated for the τ-automaton shown in Listing 4: the ζ_{push1} property is related to the *pushConstant*1 action, the ζ_{push2} property is related to the *pushConstant*2 action and the ζ_{push3} property is related to the *pushConstant*3 action.

$Formula\_Constant1\_push$
$\zeta_{push1} = \mu X.\langle invokegetExternalStorageDirectory\rangle\ \zeta_{push12} \vee$
$\langle -invokegetExternalStorageDirectory\rangle\ X$
$\zeta_{push12} = \mu X.\langle pushConstant1\rangle\ \mathtt{tt} \vee$
$\langle -pushConstant1\rangle\ X$
$Formula\_Constant2\_push$
$\zeta_{push2} = \mu X.\langle invokegetExternalStorageDirectory\rangle\ \zeta_{push22} \vee$
$\langle -invokegetExternalStorageDirectory\rangle\ X$
$\zeta_{push22} = \mu X.\langle pushConstant2\rangle\ \mathtt{tt} \vee$
$\langle -pushConstant2\rangle\ X$
$Formula\_Constant3\_push$
$\zeta_{push3} = \mu X.\langle invokegetExternalStorageDirectory\rangle\ \zeta_{push32} \vee$
$\langle -invokegetExternalStorageDirectory\rangle\ X$
$\zeta_{push32} = \mu X.\langle pushConstant3\rangle\ \mathtt{tt} \vee$
$\langle -pushConstant3\rangle\ X$

This process is repeated for each push action belonging to the PUT τ-automata considered. When all the properties are generated for all the push actions of all the τ-automata, the properties are stored in a file (*write generated properties* in the flowchart in Figure 2). Subsequently, the model checker with the PUT and GET *t-automata* and the properties obtained from the *Colluding Properties Generation* step in Figure 1 is invoked: the properties are first verified on the PUT *t-automata* and the formulae resulting TRUE are then checked with the GET *t-automata*: if the same property is resulting TRUE also on the GET τ-automata there is a collusion. In this way, we found that the colluding attack is possible between the PUT and GET *t-automata* i.e., between the (two or more) applications, whose PUT and GET models result in TRUE to the automatically generated property.

Once we explained how the proposed approach works, below we present an example to better understand how the CCS automaton are built and how the automatic properties can detect the colluding attack. We consider two real-world applications performing a colluding attack by exploiting the *ExternalStorage*, in particular we consider a *GET* application (identified by the $TZ35AO8L2KH3CZ0ILUCFMXLKP15N97M4$ hash) and a *PUT* application (identified by the $VYFLWYF4J0SHZ19DIVWSTB1CKMQLMS38$ hash).

In Figure 3, we show the bytecode related to the Android application performing the *PUT* with the *ExternalStorage* (1-Bytecode PUT in Figure 3), the CCS automaton generated by this bytecode snippet (2-CCS PUT in Figure 3) and the related τ-automaton (indicated as 3-CCS τ-PUT in Figure 3). The instructions in the bytecode performing the *PUT ExternalStorage* are in lines 146 (the invocation for the SD card access) and 151 (the resource used fro data writing) and are translated with the processes in lines 4 and 8 of the CCS automa, as shown from Figure 3.

Figure 4 shows the properties automatically generated. In particular, coherently with the formulae explained in Table 7, for each *push* actions a formula is generated. In the method related to the *PUT ExternalStorage* behaviors there are push actions related to following variable: $HKABGLTZGMM4H2M$, 2500, r, $FilesizeDdueGB$, $Collusion$, $Dataread$, and $POSTrequest$.

Figure 3. Snippet belonging to the ExternalStorage *PUT* application.

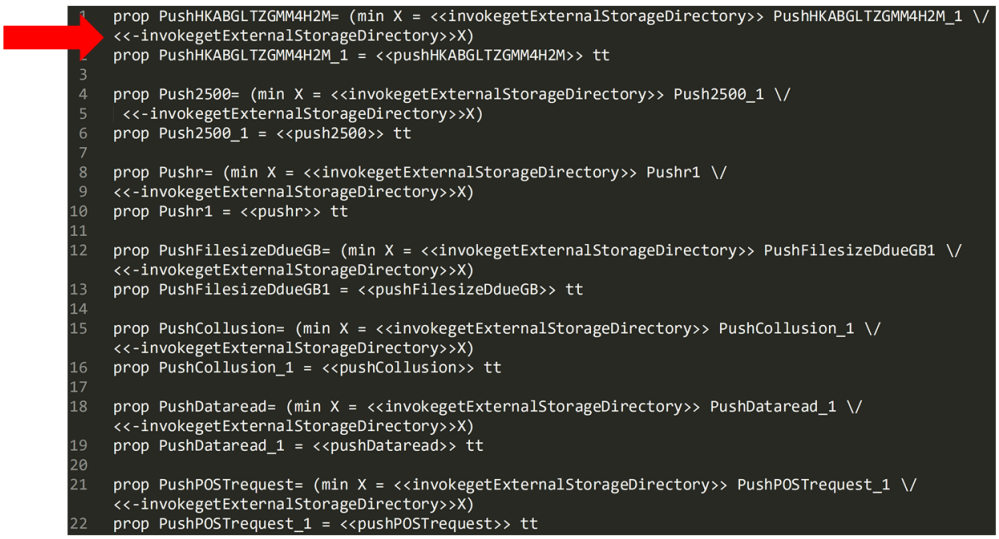

Figure 4. An example of properties automatically generated.

As shows from Figure 4 for each *push* action a property is generated. These properties, automatically generated, are evaluated on the *PUT* τ-automaton and, whether a property is resulting *TRUE*, we consider this *PUT* application as candidate to perform a *PUT* in a colluding action. In this case, the property resulting in *TRUE* by the model checker is the one marked by the red arrow in Figure 4.

Once the *PUT* model is detected, with the relative property, able to potentially perform a collusion, in order to verify whether the collusion can be successfully perpetrated, there is the need to find the *GET* application. Figure 5 shows a snippet belonging to an Android application performing a *GET* operation on an *ExternalStorage*. In particular, we show the bytecode related to the Android application performing the *GET* with the *ExternalStorage* (1-Bytecode GET in Figure 5), the CCS automaton generated by this bytecode snippet (2-CCS GET in Figure 5) and the related τ-automaton (indicated as 3-CCS τ-GET in Figure 5).

Figure 5. Snippet belonging to the ExternalStorage *GET* application.

To detect whether a collusion attack is possible, we have to invoke the model checker and verify whether the property resulting *TRUE* on the *PUT* τ-automaton is also resulting *TRUE* on the *GET* τ-automaton: this is symptomatic of a collusion attack, in this case, based on the *ExternalStorage*, which is perpetrated between these two applications.

Once collusions are detected, the proposed method shows a report as an output, where we indicate the applications involved in the collusion, the classes and methods of these applications that are related to the collusion, the name of the shared variable (i.e., the push action) and the kind of collusion attack (i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, and/or *RemoteProcedureCall*).

4. Experimental Analysis

In this section, we present the experimental analysis we performed to evaluate the effectiveness of the proposed method.

4.1. The Real-World Android Data-Set

We built a data-set by exploiting several well-known Android application repositories, composed by malicious and legitimate Android application. In particular, for the evaluation of the effectiveness of the proposed method we take into account a set of malicious applications able to perform the *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, and the *RPC* colluding attacks.

We consider the repositories to built the evaluated data-set as follows: the first one is represented by the *ACE* data-set, proposed by researchers in [10]. In this paper, the authors designed a framework to generate Android colluding applications. The ACE data-set is composed of 482 different colluding applications considering the *SharedPreferences* (160 applications), the *ExternalStorage* (160 applications), and *BroadcastReceiver* attacks (162 applications). In particular, each colluding attack, composed of 160 (162 for *BroadcastReceiver*) applications, considering 80 (81 for *BroadcastReceiver*) applications that perform a write on the shared resource (i.e., the PUT) and the remaining 80 (81 for *BroadcastReceiver*) perform a read on the same resource (i.e., GET). The second repository we considered was composed of 20 applications and developed by authors in [18]. Considering that the ACE data-set considered *SharedPreferences* colluding applications sharing only string variables, in this data-set, 10 applications exploit a collusion attack through an *Int* value, whereas the other 10 applications consider the collusion attack through a *Float* value.

With regard to the *RPC* colluding attack, we developed a set of 20 applications (10 to send data, and the remaining 10 to receive the data) by exploiting the Remote Method Procedure. As a matter of fact, Android integrates a lightweight mechanism for Remote Procedure Calls (RPCs), where a method is called locally (for instance in an Android Activity), but executed remotely (in another process, belonging for instance to an Android Service), with any result returned back to the caller. This entails decomposing the method call and all its attendant data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, and reassembling and reenacting the call there. Return values have to be transmitted in the opposite direction. Android provides all the code to do that work, and in this way, the developer can concentrate on defining and implementing the *RPC* interface itself. In the apps we developed, we consider the exchange of resources through between an Activity (i.e., an application component that provides a screen with which users can interact in order to do something) and a Service (an application component that can perform long-running operations in the background, not to provide a user interface). For data exchanging, we consider Bundles, used for passing data between various Android activities and services. The 10 applications with the Activity were able to send data (by performing a *PUT* operation on a Bundle object), whereas the 10 applications with the Service were to able to receive the data sent (by performing a *GET* operation on a Bundle object).

With regard to the non-colluding applications, we considered the *DroidBench* 2.0 (https://github.com/secure-software-engineering/DroidBench) repository, composed of 120 applications. This data-set did not have colluding applications, but some applications use *SharedPreferences*. This data-set is usually considered for the evaluation of taint analysis tools. Moreover, in order to evaluate the effectiveness of the proposed method in the detection of colluding attacks only, we took into account the *Drebin* malware repository, a repository of malicious Android applications [29,30] that do not perform collusion attacks. We chose 10 malware belonging to the top 10 malicious families in the data-set for a total of 100 non-colluding malware, as shown in Table 8.

Table 8. Malware families in the *Drebin* data-set. We selected the 10 most populous families in this repository, for a total of 100 non-colluding malware (10 samples for family). We indicate in the *Inst.* column the payload delivering (standalone, repackaging, and update), in the *attack* column the kind of attack (trojan and 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

We also considered, as malicious samples, a set of 50 ransomware samples, obtained from the Android Malware Dataset (AMD) (http://amd.arguslab.org/) repository: in detail, we considered samples belonging to the Simplelocker malicious family, whose payload is able to encrypt files with several extensions on the device external storage and then demand a ransom (usually in bitcoin) to decrypt the data.

We also considered a set of 300 trusted applications obtained from the official store of Google i.e., the *Play* store. These applications were automatically collected from Google Play (https://play.google.com/store), by using a script to query and to download applications from Android official market. In order to confirm the trustworthiness of these 260 applications, we analyzed the data-set by exploiting the VirusTotal service (https://www.virustotal.com/). This service runs 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.

To summarize, we consider a total of 1092 applications, 792 (colluding and non-colluding) malicious applications and 300 legitimate applications. The data-sets considered are composed of different types of attacks analyzed in this paper, these attacks can be colluding or non-colluding. The applications were analyzed in pairs: an application with properties of PUT was analyzed with an application with properties of GET, by going to check whether there is a collusion. In Table 9, we can see how the applications that communicate with each other through the use of Android resources were divided, so as to have a clearer picture of which applications our Model Checker has classified as colluding.

Table 9. We find the three types of attack treated in this work, and for each of them we have the number of PUT applications and the number of GET applications. In particular, for the *SharedPreferences*, we report the division of the application number by type of resource.

Type of Attack	PUT	GET
SharedPreferences	90 (80 String, 5 Int, 5 Float)	90 (80 String, 5 Int, 5 Float)
ExternalStorage	80	80
BroadcastReceiver	81	81
RemoteProcedureCall	10	10

4.2. Results

In this section, we present the results of the evaluation. To measure the performance of our approach we considered four different metrics: Accuracy, Recall, Measure F, and Accuracy.

With precision, we calculated the proportion of the examples that really belong to class X compared to all those that have been assigned to the class. It is characterized by the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved:

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

in the formula, we can see some values, such as: tp, which indicates the number of true positives and fp, which indicates the number of false positives.

The recall has been calculated as the proportion of examples assigned to class X, among all the examples that effectively belong to the class, i.e., how much part of the class was captured. Recall considers the relationship between the number of relevant records retrieved to the total number of relevant records:

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

here, we can see the value fn, which indicates the number of false negatives.

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

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

The Specificity is used to measure the proportion of negatives that are correctly identified. It is computed as follows:

$$Specificity = \frac{tn}{tn + fp}$$

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 (true positive, false negative, false positive and true negative):

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

as already specified in the parenthesis tn indicates the number of true negatives.

In Table 10 we show the obtained results.

Table 10. Performance results.

Precision	Recall	F-Measure	Specificity	Accuracy
1	1	1	1	1

From the experimental analysis results shown in Table 10, we obtain an accuracy and a specificity equal to 1, showing that the proposed method is able to correctly classify all the colluding applications without any misclassifications with malware not performing colluding actions and legitimate samples. In particular, we were able to correctly classify colluding applications because the name of the resource shared is the same between the GET and the PUT application because i.e., a *push* action.

5. State-Of-The-Art Literature

The anti-malware detection techniques actually converge their attention on the analysis of one sample at a time, which excludes the presence of communication channels between the applications.

The research community is working on Inter-Component Communication to develop innovative tools and methods to identify these channels.

A particular kind of colluding attack is characterized by *covert channel*; the authors in [31], developed a multichannel communication mechanism to transfer sensitive data securely on mobile devices, called the Multichannel Communication System (MSYM). It uses the VpnService interface (https://developer.android.com/reference/android/net/VpnService) provided by the Android platform, and thus is able to intercept the network data sent and then split it into different parts that will be disordered and encrypted through multiple transmission channels.

The accelerometer sensor represents a type of covert channel and is able to generate signals that reflect users' motions. These data can be read by malicious applications, but correctly using the device's vibration engine, the stolen data can be encrypted, so one application can cause acceleration data to be received and decrypted by another application. For this reason, two Android applications (representing the source and the sink, respectively) were developed and [32] tested on three different smartphone models to verify this type of communication.

Usually the covert channels are divided into the covert storage channel and covert timing channel [33]. The difference is that:

- the covert storage channel hides the secret information in the data transmission protocol and achieves covert communication using payload or non-payload fields:
- the covert timing channel encodes the secret information and achieves the covert communication using the time gap between packets.

There are many methods able to detect specific covert timing channels, but the authors in [34] worked to perform the filtering, grouping, and other operations regarding the communication packets. They also chose a suitable machine learning algorithm to train the classification model, obtaining good detection performance.

The magnetometers, better known as magnetic sensors found in devices such as smartphones and tablets are typically used for positioning and orientation. They can be exploited by the attackers to exfiltrate information from isolated and non-networked computers. MAGNETO [35] is a proposal of covert communication about air-gapped systems and nearby smartphones via magnetic fields generated from the CPU. The magnetic signals can be generated from the computers changing the CPU workload, but this covert channel works only for short distances and with low transmission rate. It is effective in unconstrained environments having the wireless communications blocked.

TaintDroid [1] is an extension of the Android operating system and specifically tracks the flow of privacy-sensitive data passing through third-party applications. In most cases, downloaded third-party applications are not reliable, which is why the approach monitors, in real time, how these applications access users' personal data and manipulate them.

Inter-Component Communication based Taint Analysis tool (IccTA) [36] is a tool that uses the static taint analysis technique, its goal is to recover paths where privacy and sensitive information are sent outside without users being aware and therefore without their permission. The approach is able to detect paths within a single component or between multiple components. For testing IccTA, the developer investigated 22 applications containing ICC-based privacy leaks.

Concerning the IccTA, the researchers in [37] have developed Amandroid, a tool that focuses its attention on the detection of leaks via an ICC analysis. The execution of the ICC analysis involves the generation of two components, called Inter-Component Data Flow Graph (ICDFG) and Data Dependence Graph (DDG), respectively. For each component, it performs a data flow and data dependency analysis, and also tracks the communication exchange between them, providing a customized analysis on Android applications.

An approach called DroidSafe [38] performs static information flow analysis, reporting any sensitive data leaks from Android applications. The Soot Java Analysis Framework was used to

develop DroidSafe and it works by analyzing one application at a time, for this reason, it is not able to detect a collusion attack that is caused by two or more applications.

Another methodology for the inter-application communication threats detection is MR-Droid [39]. It is focused, in particular, on intent spoofing and collusion, and to work uses a framework based on MapReduce to execute a compositional application analysis.

The authors of SneakLeak [40] developed a new tool for detection of application collusion, based on model checking. It works by analyzing multiple applications simultaneously and is able to identify set of suspicious applications that may be involved in a collusion attack. The tests were executed on a set of Android applications belonging to the DroidBench data-set, which show collusion through inter-application communication.

In [41], researchers developed a tool to inspect the manifest file of an APK to dynamically detect an application collusion. It identifies the permission request with respect to the permission shown to the user during installation and it executes a dynamic check on the APIs used within the applications and with which other applications the original APK is communicating.

AppHolmes [42] is a static analysis tool to detect app collusion. This tool extracts the manifest and the smali code from the APKs. Then, it performs a static analysis on the smali code, identifying possible Intent values from all callsites of startService(), bindService(), and sendBroadcast(). These callsites can cause collusion with the corresponding Intents. Finally, it execute a match between all detected Intents and IntentFilters obtained with the manifest files. If an Intent from app A matches to an IntentFilter from app B, it is reported a collusion by AppHolmes.

UpDroid [43] is a data-set entirely composed by update attacks. It contains 21 families and 2479 samples. This data-set consists of malicious applications that use updating techniques in order to elude the detection systems. In an update attack, the original application is considered benign, so in not the installation of the application to launch the attack. The malicious payload is introduced during the application update. To address this problem, the authors in [44] propose a technique that applies formal methods for the update attack identification, which are able to localize the code portion that implements the download. The tool transform the APKs in CCS file and then translates the Java bytecode into CCS process specifications, which will be transformed into formal models.

The formal methods are also used by the authors in [45], where they propose a technique based on formal methods to detect ransomware on smartphone devices through the Java bytecode. Instead, the authors in [46] propose another approach based on gathering information about system behavior using a virtual machine for the execution. If there is a Python script on the machine, it is likely that the script and its generation could be destroyed. When the program obtains information about the process, it sends it to a monitor via a secure connection channel. The monitoring systems is based on Linux in order to minimize ransomware infection.

6. Conclusions and Future Work

The spread of smartphones has allowed the creation of a new target for cybercriminals, who are committed to constantly creating new malware in order to evade controls and gain access to people's sensitive data. In our study, investigated the colluding attack, which is based on the communication between two or more malicious applications.

Our tool works by using formal methods to detect the presence of collusion. An automaton is generated for each class of an Android application by transforming the java bytecode instructions into CCS processes. With the model checking technique, we verify if the automata satisfy the pre-filtering properties, in this way, we obtain the set of candidate colluding applications. We then generate the τ-automata of the candidate applications, containing the push actions useful to identify the different type of colluding attacks (*SharedPreferences*, *ExternalStorage*, *BroadcastReceiver*, and RPC), to understand if we have detected any applications capable of collusion.

To test the performance of our method, we considered several data-sets with a total of 1092 applications, divided as follows: the ACE data-set (482 applications, all colluding), DroidBench 2.0

(120 applications, no colluding), Drebin malware repository (100 non-colluding malware), [18] repository (40 applications (10 *SharedPreferences*-Int, 10 *SharedPreferences*-Float, 20 *RemoteProcedureCall)*, all colluding), AMD repository (50 ransomware sample, no colluding), and a set of legitimate applications from Google Play Store (300 applications, no colluding).

As future work, we plan to apply the proposed method on an extended set of colluding attacks, with the aim of evaluating the proposed methods effectiveness in the detection of new colluding attacks. For this reason, we plan to develop a framework to automatically inject colluding malicious payloads into legitimate Android applications, by extending the functionalities of the tool proposed by authors in [10]. Moreover, as an additional research line, we plan to investigate whether it is possible to reduce the actions of automata to apply formal equivalence checking for colluding detection. One of the limitations of our method is that the generation of the first heuristic is not automatic. In fact, new collusion attacks require the security analyst to first formulate a formula for the identification. The first formula is used as a heuristic to filter all classes that perform *GET* and *PUT* actions, for this reason, the method can also work by considering only the second heuristic (the automatic property generation); however, more computational time will be required.

Author Contributions: Conceptualization, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) and A.S.; methodology, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi), A.S.; software, R.C., F.M. (Francesco Mercaldoi); validation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) and A.S.; formal analysis, R.C., F.M. (Francesco Mercaldoi) and A.S.; investigation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) and A.S.; resources, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) and A.S.; data curation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) and A.S.; writing–original draft preparation, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) and A.S.; writing–review and editing, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) and A.S.; visualization, R.C., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldoi) 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, EU SPARTA, CyberSANE and E-CORRIDOR projects.

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

References

1. 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]
2. Arabo, A.; Pranggono, B. Mobile malware and smart device security: Trends, challenges and solutions. In Proceedings of the 2013 19th International Conference on Control Systems and Computer Science, Bucharest, Romania, 29–31 May 2013; IEEE: New York, NY, USA, 2013; pp. 526–531.
3. Enck, W. Defending users against smartphone apps: Techniques and future directions. In Proceedings of the International Conference on Information Systems Security, Kolkata, India, 15–19 December 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 49–70.
4. Arabo, A.; Brown, I.; El-Moussa, F. Privacy in the age of mobility and smart devices in smart homes. In Proceedings of the 2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing, Amsterdam, The Netherlands, 3–5 September 2012; IEEE: New York, NY, USA, 2012; pp. 819–826.
5. 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, Maui, HI, USA, 7–10 January 2020
6. 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]
7. 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]

8. 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; IEEE: New York, NY, USA, 2016; pp. 744–746.
9. 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]
10. Blasco, J.; Chen, T.M. Automated generation of colluding apps for experimental research. *J. Comput. Virol. Hacking Tech.* **2018**, *14*, 127–138. [CrossRef]
11. Xu, K.; Li, Y.; Deng, R.H. Iccdetector: Icc-based malware detection on android. *IEEE Trans. Inf. Forensics Secur.* **2016**, *11*, 1252–1264. [CrossRef]
12. 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.
13. 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; IEEE: New York, NY, USA, 2019; pp. 5998–6000.
14. Feng, H.; Shin, K.G. Understanding and defending the Binder attack surface in Android. In Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles, CA, USA, 5–9 December 2016; pp. 398–409.
15. Gajrani, J.; Agarwal, U.; Laxmi, V.; Bezawada, B.; Gaur, M.S.; Tripathi, M.; Zemmari, A. EspyDroid+: Precise reflection analysis of android apps. *Comput. Secur.* **2020**, *90*, 101688. [CrossRef]
16. Memon, A.M.; Anwar, A. Colluding apps: Tomorrow's mobile malware threat. *IEEE Secur. Priv.* **2015**, *13*, 77–81. [CrossRef]
17. 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.
18. Casolare, R.; Martinelli, F.; Mercaldo, F.; Santone, A. Android Collusion: Detecting Malicious Applications Inter-Communication through SharedPreferences. *Information* **2020**, *11*, 304. [CrossRef]
19. 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.
20. Milner, R. *Communication and Concurrency*; PHI Series in Computer Science; Prentice Hall: Upper Saddle River, NJ, USA, 1989; pp. 1–11.
21. Stirling, C. An Introduction to Modal and Temporal Logics for CCS. In *Concurrency: Theory, Language, and Architecture*; Springer: Berlin/Heidelberg, Germany, 1989; pp. 2–20.
22. De Francesco, N.; Lettieri, G.; Santone, A.; Vaglini, G. Heuristic search for equivalence checking. *Softw. Syst. Model.* **2016**, *15*, 513–530. [CrossRef]
23. Altuwaijri, H.; Ghouzali, S. Android data storage security: A review. *J. King Saud Univ.-Comput. Inf. Sci.* **2020**, *32*, 543–552. [CrossRef]
24. Kumar, S.; Shukla, S.K. The State of Android Security. In *Cyber Security in India*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 17–22.
25. Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A.; Sangaiah, A.K.; Cimitile, A. Evaluating model checking for cyber threats code obfuscation identification. *J. Parallel Distrib. Comput.* **2018**, *119*, 203–218. [CrossRef]
26. Bacci, A.; Bartoli, A.; Martinelli, F.; Medvet, E.; Mercaldo, F. Detection of obfuscation techniques in Android applications. In Proceedings of the 13th International Conference on Availability, Reliability and Security, Hamburg, Germany, 27–30 August 2018; pp. 1–9.
27. Bacci, A.; Bartoli, A.; Martinelli, F.; Medvet, E.; Mercaldo, F.; Visaggio, C.A. Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis. In Proceedings of the ICISSP, Madeira, Portugal, 22–24 January 2018; pp. 379–385.
28. De Lorenzo, A.; Martinelli, F.; Medvet, E.; Mercaldo, F.; Santone, A. Visualizing the outcome of dynamic analysis of Android malware with VizMal. *J. Inf. Secur. Appl.* **2020**, *50*, 102423. [CrossRef]
29. 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, 24–28 August 2015; IEEE: New York, NY, USA, 2015; pp. 333–340.

30. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. *Ndss* **2014**, *14*, 23–26.
31. 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]
32. Al-Haiqi, A.; Ismail, M.; Nordin, R. A new sensors-based covert channel on android. *Sci. World J.* **2014**, *2014*, 969628. [CrossRef]
33. Shrestha, P.L.; Hempel, M.; Rezaei, F.; Sharif, H. A support vector machine-based framework for detection of covert timing channels. *IEEE Trans. Dependable Secur. Comput.* **2015**, *13*, 274–283. [CrossRef]
34. Han, J.; Huang, C.; Shi, F.; Liu, J. Covert timing channel detection method based on time interval and payload length analysis. *Comput. Secur.* **2020**, *97*, 101952. [CrossRef]
35. Guri, M. Magneto: Covert channel between air-gapped systems and nearby smartphones via cpu-generated magnetic fields. *Future Gener. Comput. Syst.* **2020**. [CrossRef]
36. 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; IEEE: New York, NY, USA, 2015; Volume 1, pp. 280–291.
37. 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]
38. Gordon, M.I.; Kim, D.; Perkins, J.H.; Gilham, L.; Nguyen, N.; Rinard, M.C. Information flow analysis of android applications in droidsafe. *NDSS* **2015**, *15*, 110.
39. 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, 25–25 May 2017; IEEE: New York, NY, USA, 2017; pp. 189–198.
40. Bhandari, S.; Herbreteau, F.; Laxmi, V.; Zemmari, A.; Roop, P.S.; Gaur, M.S. Sneakleak: Detecting multipartite leakage paths in android apps. In Proceedings of the 2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, 1–4 August 2017; IEEE: New York, NY, USA, 2017; pp. 285–292.
41. Arabo, A. Mobile App collusions and its cyber security implications. In Proceedings of the 2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud), Beijing, China, 25–27 June 2016; IEEE: New York, NY, USA, 2016; pp. 178–183.
42. Xu, M.; Ma, Y.; Liu, X.; Lin, F.X.; Liu, Y. AppHolmes: Detecting and characterizing app collusion among third-party Android markets. In Proceedings of the 26th International Conference on World Wide Web, Perth, Australia, 3–7 April 2017; pp. 143–152.
43. Aktas, K.; Sen, S. Updroid: Updated android malware and its familial classification. In Proceedings of the Nordic Conference on Secure IT Systems, Oslo, Norway, 28–30 November 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 352–368.
44. Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Download malware? no, thanks: How formal methods can block update attacks. In Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, Austin, TX, USA, 15 May 2016; pp. 22–28.
45. Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Ransomware steals your phone. Formal methods rescue it. In Proceedings of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems, Heraklion, Greece, 6–9 June 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 212–221.
46. Arabo, A.; Dijoux, R.; Poulain, T.; Chevalier, G. Detecting Ransomware Using Process Behavior Analysis. *Procedia Comput. Sci.* **2020**, *168*, 289–296. [CrossRef]