Identifying Insecure Features in Android Applications using Model Checking

Fabio Martinelli[1], Francesco Mercaldo[1] and Vittoria Nardone[2]
[1]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy*
[2]*Department of Engineering, University of Sannio, Benevento, Italy*

Keywords: Android, Model Checking, Security.

Abstract: Nowadays Android is the most widespread operating system. This is the reason why malware writers target it. Both researchers and commercial antimalware provide several solutions to fix and detect this phenomenon. They analyze one single application per time using combinations of static, dynamic and behavior based techniques. However, one of the last new threats is the collusion attack. In order to perpetrate this attack the malicious behaviour is divided between two or more applications: collusion refers to multiple applications that accomplish their fragment of malicious behaviour and then communicate using the Inter Component Communication mechanism provided by Android platform. Basically the colluded applications intentionally put in view private and sensitive information. The aim of this paper is to investigate whether legitimate and malware applications share private data. One way to exchange data between different applications in Android environment is through Shared Preferences. In this preliminary work we investigate whether an application transfers data using Shared Preferences with public visibility.

1 INTRODUCTION

As reported in the IDC report[1], Android continues to be the most dominant platform and its popularity will continue to grow. Due to this large popularity, malware writers are developing more aggressive paradigms of attack.

The new trend in this area is represented by the so-called colluding attack: the malicious action is splitted in several applications. In this way current antimalware technologies, that analyze an application in order to discover harmful action as a single entity, are not able to identify the malicious payload.

In collusion attacks, a malicious operation is performed through multiple applications: it is divided into multiple parts and distributed among a set of colluding applications. In order to achieve their malicious aims, the applications communicate trough Inter Component Communication (ICC) mechanism. Each application avoids suspicion by requesting the minimum permissions needed for its role. For example, when two applications collude, the first application might read sensitive data and transmit them to the second application, which transmits them to the outside world. Analyzed individually, the applications would be considered benign because there is no direct path from sensitive data to their transmission.

Different research works ((Xu et al., 2016; Li et al., 2014; Enck et al., 2014; Wei et al., 2014)) investigate information leaks using inter-component data flow analysis. However, these security techniques are focused to detect vulnerabilities in a single application, but fail to identify vulnerabilities involving multiple applications.

Recently, academic researches have extended flow analysis to include multiple applications using ICC channels in order to communicate with each other (Li et al., 2015b; Bhandari et al., 2017).

Conversely, this paper wants to investigate whether legitimate and malware applications intentionally put in view some data. In fact, in colluding attack the applications intentionally exhibit private data. Thus, this work not performs data flow analysis but it investigates whether a data vulnerability is exposed using model checking.

The proposed method aims to identify Shared Preferences feature with a public data, since intentionally sharing some data is the first step to mark an application as good candidate in order to exfiltrate data.

The remainder of this paper is organized as fol-

[1]https://www.idc.com/getdoc.jsp?containerId=prUS41425416

Martinelli F., Mercaldo F. and Nardone V.
Identifying Insecure Features in Android Applications using Model Checking.
DOI: 10.5220/0006758105890596
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 589-596
ISBN: 978-989-758-282-0

lows. In Section 2 the state of the art related to colluding attack research effort is analyzed. Section 3 provides the background knowledge required to understand the working environment. Section 4 motivates the aim of the work through an illustrative example. In Section 5 the workflow of the proposed methodology is depicted. Section 6 shows the results obtained during the evaluation and Section 7 discusses some aspect of the work and investigates for further directions. Finally, Section 8 concludes the work.

2 RELATED WORK

The following section presents different experimental approaches for the detection of intra and inter-application communication vulnerabilities.

Authors in (Liu et al., 2017) present MR-Droid a MapReduce based framework able to detect inter-app communication threats. Is is specialized in intent hijacking, intent spoofing and collusion. MR-Droid builds a large-scale ICC graph extracting data-flow features between multiple communicating apps. This kind of approach is scalable and accurate. However, it is able to only handle intent based communications. Another limitation is related to detection of privacy leakage across two apps.

Bhandari et al. in (Bhandari et al., 2017) present a model-checking based approach to detect inter-app collusion. The proposed approach is composed by four steps. The first step aims to collect information related to ICC sources and sinks. The second step carries out data-flow analysis in order to map sensitive information provided by sensitive API call. The third step aims to generate for each app a PROMELA model. In the fourth step, model checking is applied in order to verify the collusion detection property on the generated models. Whether the compositional model of apps satisfy the property, then that set of apps is not a colluding set candidate.

Filiol and Irolla in (Filiol,) propose three different tools. The first one, called Tarentula, is a web crawler collecting mobile applications. The second one, called Egide, is a static malware detector. Finally Panoptes, the third one, is an advanced dynamic tool analyzing network communications. It can monitor all communications based on HTTP, HTTPS, POP, IMAP, SMTP between the application and the Internet. It especially addresses the issue of data privacy regarding banking apps and Facebook app. (Filiol,) analyzes only the network traffic, instead this work aims to investigate the feasible exposure of some data using the Android communication channels.

Authors in (Asavoae et al., 2016) propose two approaches able to identify collusion apps candidates. The first one is a rule based approach developed in Prolog. The second one is a statistical based approach. The rule based approach use some features (e.g. permissions, communication channels) to identify colluding apps. In the statistical approach a probabilistic model is defined. Additionally the authors say that also model-checking is a feasible approach to detect collusion in Android apps.

Authors in (Li et al., 2015b) propose IccTA which is a static taint analyzer to detect privacy leaks between components in Android applications. Combining it with APKCombiner (Li et al., 2015a), IccTA can also detect inter-app leakage paths. IccTA goes beyond state-of-the-art approaches by supporting inter-component detection. IccTA improves the precision of the analysis spreading context information between components.

In (Bagheri et al., 2015) Bagheri et al. present COVERT, a tool for compositional analysis of Android inter-app vulnerabilities. COVERT statically analyzes the source code of each application, then it extracts relevant security specifications in a format suitable for formal verification. COVERT uses model checker tool in order to verify whether the formal model defined is affected by collusion.

Conversely from the above approaches, this work aim to investigate whether an application defines public communication channels. In particular, this paper investigates if into applications there are Shared Preferences with a world readable visibility. As specified in the Android developers guide, using a Shared Preferences with this kind of visibility is very dangerous. In these kind of cases every applications have access to the Shared Preferences information.

3 PRELIMINARIES

This section aims to introduce some preliminary concepts related to the android environment and model checking technique.

3.1 Android Overview

Linux Kernel is the base of the Android Operating system. In Android system the applications run in an isolated sandbox. In order to protect the device the Android operating system provides several security features: application signing, application-defined and user-granted permissions and secure interprocess communication. Permissions are used to regulate the access to sensitive resources, such as users' personal information or sensor data (e.g., camera, GPS, etc.).

Android applications are developed in Java. They are composed by four different components: Activities, Services, Broadcast Receivers and Content Providers. An application may be build using many different components and any their combination. In the following, these components are briefly described:

An **Activity** usually allows the interaction with the physical user. It includes a set of GUI components representing the visible interface to the user.

A **Service** deals of background computations which keep on even when the application is not in focus. It also provides interfaces for Inter Component Communication exposing some of its functionality to other applications.

Broadcast Receivers allow to receive broadcast messages both from the other application or the system. It implements a sort of publish-subscribe design pattern. Usually, when an event occurs a broadcast is sent by either the system or an application. In order to receive specific broadcasts an application can be subscribed to it.

Content Provider provides a mechanism able to sharing data with other applications. It shares data as relational database-like model.

3.1.1 Inter Component Communication (ICC)

Android framework provides some channel for inter component communication. The main Android communication channels are the following:

- **Intent** allows to application components to invoke other components of the same or other applications. Using Bundles it is also possible to pass data between different components. An Intent is the preferred messaging object for asynchronous ICC in Android. It contains the following primary information: component name, action string, category and data. These information are optional. Based on presence/absence of component name (i.e. the destination) there are two type of intent: Explicit and Implicit Intent. In the first case the destination is specified in the second case no destination is specified.
- **Content Provider** is like relational databases. It stores structured data and can be used to transfer data across components of same or different apps. The data can be accessed and modified using ContentResolver objects. Best practice is to attach read and write permissions to the content provider, in order to protect it from data exposure.
- **External Storage** is the storage space external to an app. It is placed in the SD card. External storage can be accessed using the following permissions: `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE`.
- **Shared Preferences** allow applications to store key-value pairs of data. It is an operating system feature and it is usually used to store preferences information. Applications can access to key-value pairs data according to the mode visibility defined for the preferences. This paper aim to focus on mode visibility used to define a Shared Preferences. Since using a public visibility could be expose an application to a data exfiltration. To more detail see Section 4.

3.2 Model Checking

Complex systems verification and specification is usually performed using Formal Methods. Model Checking belongs to formal techniques and requires three main phases:

- modeling the system with a precise notation;
- specifying the system behavioural properties with a precise notation;
- verifying the properties on the system using a model checker tool.

While model checking was originally developed to verify the correctness of systems, recently it has been also proposed in other fields such as clone detection (Santone, 2011), biology (De Ruvo et al., 2015), secure information flow (De Francesco et al., 2003), and mobile computing (Anastasi et al., 2001). In the last years, model checking has been successfully applied also in the security field, as explained in the following works: (Battista et al., 2016) and (Mercaldo et al., 2016). The main aim of these works is to detect and identify Android malware families.

In the following the three phases of model checking are detailed.

Modeling the System

The system behavior is defined using an automaton. It has a set of labeled edges and a set of nodes. The states of the system are represented by the nodes while the transitions from a state to another state (precisely the next state) are represented by the edges. Every edges have a label meaning that the system can switch from a state s to a state s' performing an action a. This action a is the label of the edge. The automaton' root is the initial state of the system and usually a transition is depicted as follow: $s \xrightarrow{a} s'$.

Processes are an algebraical helpful representation to define the automaton. Thus, the precise notation used to describe complex systems is usually the process algebra. The methodology of this work uses Milner's Calculus of Communicating Systems (CCS)

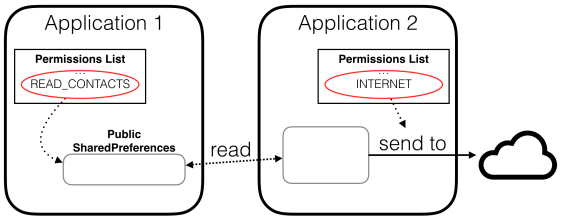

Figure 1: Vulnerable Applications: a Simple Example.

(Milner, 1989) as process algebra. In CCS process algebra the states of the system are modeled through processes and the transitions are modeled using the actions. The reader can refer to (Bruns, 1997; Milner, 1989) for more details on CCS.

Specifying the Properties

The behavioural properties of the system can be defined using a temporal logic. Usually properties are sequences of actions that a system should perform in order to accomplish a particular purpose. In order to formally verify this kind of properties temporal logic constructs are used. General examples of properties can be the following: verifying whether a particular event will eventually occur or a property is verified in every state. This work uses the *mu-calculus* logic(Stirling, 1989).

Formal Verification Environment: Model Checker Tool

The last phase of Model Checking technique is the verification of the specified properties (temporal logic formulae) on the system (labeled transition system). To accomplish this aim a model checker tool is usually used. It takes as input both the modeled system and the properties. The model checker provides a binary output: whether the system verifies the property it return `TRUE` or `FALSE` otherwise. The tool performs an exhaustive state space search is guaranteed to terminate since the model is finite. In this work the *Concurrency Workbench of New Century (CWB-NC)* (Cleaveland and Sims, 1996) is used as formal verification environment.

4 MOTIVATING EXAMPLE

Figure 1 shows a pattern example of inter applications vulnerabilities. This example point out the privilege escalation that motivates this work. The figure shows a malicious application **App1** sharing a public Shared Preferences containing the contact list of a user. **App2** takes this list, without the `READ_CONTACTS` permission, and sends it, through Internet, to a remote server. This is a simple example of a vulnerability using Inter Component Communication (ICC) among Android apps.

As the figure shown, the privilege escalation is exploited since the Shared Preferences visibility is public. For this reason, our work aims to investigate in Android environment whether legitimate and malware applications employ Shared Preferences with public visibility. Thus, this work searches the insecure feature related to a public Shared Preferences.

5 METHODOLOGY

Figure 2 shows the methodology workflow. This model checking based methodology has two main steps:

- building the model starting from Bytecode instructions of an Android application;
- specifying the properties checking insecure features related to Android Shared Preferences.

Formal Model

In order to build the formal model of an Android application this methodology aims, using a reverse engineering process, to obtain the app executable files, i.e. the `.class` files. Starting from the apk file dex2jar[2] tool is used to obtain the Java Archive file (`jar`) from the application Dalvik Executable file (`dex`). The executable files are extracted from the from the `jar` file through the `jar -xvf` command provided by the Java Development Kit. At the end of this reverse engineering process the classes of the Android application are obtained. Afterwards, Apache Commons Bytecode Engineering Library (BCEL)[3] is used to parse the Bytecode. Finally, every Bytecode instruction is translated in an algebraical process. In particular, as state before, this methodology uses CCS language (Calculus of Communicating Systems of Milner (Milner, 1989)). Using a Java Bytecode-to-CCS transformation function every Java Bytecode instruction is translated in a CCS process. This representation allows to simulate the normal flow of the instructions. A simple example of this transformation function is the following: the `if` statement is translated as a non-deterministic choice. In this case the system can evolve from a process s to two different processes s' and s'', corresponding to the two alternative paths (true/false) of the classical `if` statement.

The formal model construction is an automatic process in this methodology. The output of this first step is the formal model of the application.

[2]https://sourceforge.net/projects/dex2jar/
[3]http://commons.apache.org/bcel/

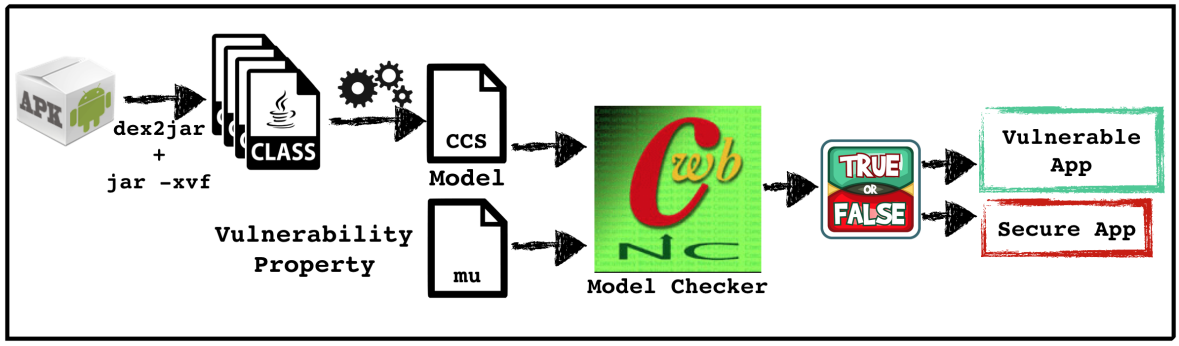

Figure 2: Methodology Workflow.

Behavioural Properties

The aim of this second step is to investigate whether an application is developed using insecure feature which may cause security issues in mobile applications. In this methodology the defined behavioural property investigates whether a Shared Preferences is declared as: `MODE_WORLD_READABLE`. Usually, this mode is indicated through the (0x00000001) constant value. In Bytecode this value is translated with the `iconst_1` opcode. In order to better understand this kind of insecure feature, an example of Shared Preferences declaration is reported in Listing 1.

As listing 1 shown, the `getSharedPreferences` method has two arguments: the name of the preferences file and the operating mode. If a preferences file by the specified name does not exist, it will be created editing and committing the changes. The operating mode can assume the following values: `MODE_PRIVATE`, `MODE_WORLD_READABLE`, `MODE_WORLD_WRITEABLE` or `MODE_MULTI_PROCESS`. Using the `MODE_PRIVATE` mode the created file can only be accessed by the calling application. The other cases allow all other applications to have access to the created file. Creating files using these ways can imply security issues, and may generate insecurity applications. For this reason this paper aims to investigate whether an application uses public Shared Preferences which are visible to all other applications installed on mobile devices.

Table 1 shows the logic rule φ related to the above explained unsafe behaviour.

The mu-calculus logic, (Stirling, 1989) as a branching temporal logic, is used to describe a determined behavior of the app. In this formula is used the last fixpoint ($\mu Z.\phi$) of the recursive recursive equation $Z = \phi$. μZ *binds* free occurrences of Z in ϕ. An occurrence of Z is free if it is not within the scope of a binder μZ. Finally, at the end of this second step, the temporal logic rule is specified.

Formal Verification Environment

In this approach the Concurrency Workbench of New Century (CWB-NC) (Cleaveland and Sims, 1996) is used as formal verification environment. The CWB-NC model checker takes as input the formal model of an application (written in CCS) and the temporal logic property written in mu-calculus. The output of this verification step is binary: `true`, whether the property is verified on the model and `false` otherwise. For this approach, an application is considered insecure whether the property φ shown in Table 1 is verified on the model.

It is well-known that a model checking technique typically suffers of the state explosion problem. In fact, it is mainly applicable to small-scale applications, but do not scale up well. However, the state explosion problem is not a real problem when verifying Android applications, since the produced CCS specifications do not have a large number of states and transitions.

6 SECURITY EVALUATION

In order to evaluate this methodology, two different dataset are used: 200 legitimate applications and 200 real world samples belonging to the Drebin project's dataset (Arp et al., 2014; Spreitzenbarth et al., 2013).

The legitimate applications are downloaded from Google Play[4], the Android official market. These trusted applications have been crawled from the Google's official app store[5] using an open-source crawler[6]. They were downloaded by between January 2016 and April 2016. The legitimate applications of this dataset belonging to all the different categories available on the market.

[4]https://play.google.com
[5]https://play.google.com/store
[6]https://github.com/liato/android-market-api-py

```
SharedPreferences getSharedPreferences(String name, int mode)
```

Listing 1: Example of SharedPreferences declaration.

Table 1: An example of logic rule.

$$\varphi \quad = \quad \mu X = \langle iconst\_1 \rangle \langle invokegetSharedPreferences \rangle \, \mathtt{tt} \vee \langle -iconst\_1, invokegetSharedPreferences \rangle \, X$$

The malware dataset is a very well known collection of malware which includes the most diffused Android: in the evaluation we consider four different families with 50 samples for each one of them. The analyzed families are the following: FakeInstaller, GinMaster, Opfake and Plankton.

Table 2 shows the results achieved during the investigation related to the legitimate applications. Table 2 is organized as follow: considering the φ formula, # **Secure Apps** is the number of apps resulting false to the formula; # **Insecure Apps** is the number of apps resulting true to the formula. The last column is the number of total applications analyzed in this preliminary work.

As the table shown, 17 applications are classified as insecure. In fact, these applications declare `Shared Preferences` with a public visibility. Creating world-readable files is very dangerous and 17 applications belonging to this dataset have done it.

Table 2: Security Evaluation on Legitimate Applications.

Family	**# Secure Apps** $\varphi = FALSE$	**# Insecure Apps** $\varphi = TRUE$	**# Legitimate Apps**
Legitimate	183	17	200

Conversely, Table 3 shows the results achieved during the investigation related to the malware samples. Table 3 is organized as follow: each row shows the results achieved by each family evaluating the φ formula. As stated before, this methodology considers as **Secure Apps** the samples resulting false to the formula φ and classifies as **Insecure Apps** the samples resulting true to the formula φ. The last column shows the number of samples belonging to each family involved in this evaluation. As the table shown, 47 malware samples are classified as insecure, this means that 47 malware samples have defined `Shared Preferences` with a public visibility. The results also highlight that only the samples belonging to Opfake and Plankton families expose this kind of vulnerability.

Another result is related to the comparison between legitimate and malware applications. In particular, the tables 2 and 3 show that the number of malware samples affected by Shared Preferences vulnerability (47) is greater than the number of legitimate applications (17). This preliminary result seems to indicate that legitimate applications are less vulnerable if compared with the malware samples, since Shared Preferences with a public visibility are most commonly used by malware samples.

Table 3: Security Evaluation on Malware Samples

Family	**# Secure Apps** $\varphi = FALSE$	**# Insecure Apps** $\varphi = TRUE$	**# Malware**
FakeInstaller	50	0	50
GinMaster	50	0	50
Opfake	22	28	50
Plankton	31	19	50
Total	**153**	**47**	**200**

7 DISCUSSION AND FUTURE EXTENSIONS

The evaluation of the proposed method consider a reduced number of legitimate and malware applications. Thus, the first improvement is related to the evaluation of an extended dataset. For example, as future work a grater number of samples have to be considered in the evaluation, including also other malware families.

Another limitation of this work is related to its ability to handle only Shared Preference vulnerability. Other kinds of vulnerability can be detected in the future, e.g. intent based ICC communications. Usually, Intents are the most common components used to inter-app communication. For example, a future work can investigate whether an intent is vulnerable, with particular attention on the implicit intent.

Furthermore, it is also interesting to investigate whether these kinds of vulnerability generate collusion attacks. For example, for the Shared Preferences vulnerability, a combiner can be built. This combiner aim to link two applications, the first application that is able to put something in a Shared Preferences and the second one able to get to the resource. Afterwards, some logic formulae can be used in order to verify if information flow has occurred.

8 CONCLUSIONS

Android is the most widespread mobile operating system. Malware writers threaten it using new types of attacks. One of the last new threats is the collusion attack. During this kind of attack the malicious behaviour is performed through multiple applications: two or more applications collaborate in order to accomplish malicious aims. In collusion attacks the applications intentionally put in view some private data. This paper aims to investigate whether legitimate and malware applications show public user data. This is a kind of vulnerability detected by the methodology designed in this work using a dataset composed by 200 legitimate applications downloaded from the Android official store and 200 malware samples belonging to the Drebin dataset. The results show that some legitimate applications intentionally expose public data. Instead, Opfake and Plankton malware samples largely expose the public Shared Preferences vulnerability.

ACKNOWLEDGEMENTS

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII.

REFERENCES

Anastasi, G., Bartoli, A., De Francesco, N., and Santone, A. (2001). Efficient verification of a multicast protocol for mobile computing. *Computer Journal*, 44(1):21–30. cited By 12.

Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., and Rieck, K. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*.

Asavoae, I. M., Blasco, J., Chen, T. M., Kalutarage, H. K., Muttik, I., Nguyen, H. N., Roggenbach, M., and Shaikh, S. A. (2016). Towards automated android app collusion detection. *arXiv preprint arXiv:1603.02308*.

Bagheri, H., Sadeghi, A., Garcia, J., and Malek, S. (2015). Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41:866–886.

Battista, P., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016). Identification of android malware families with model checking. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 542–547.

Bhandari, S., Herbreteau, F., Laxmi, V., Zemmari, A., Roop, P. S., and Gaur, M. S. (2017). Poster: Detecting inter-app information leakage paths. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 908–910. ACM.

Bruns, G. (1997). *Distributed Systems Analysis with CCS*. Prentice-Hall.

Cleaveland, R. and Sims, S. (1996). The ncsu concurrency workbench. In *CAV*. Springer.

De Francesco, N., Santone, A., and Tesei, L. (2003). Abstract interpretation and model checking for checking secure information flow in concurrent systems. *Fundamenta Informaticae*, 54(2-3):195–211. cited By 12.

De Ruvo, G., Nardone, V., Santone, A., Ceccarelli, M., and Cerulo, L. (2015). Infer gene regulatory networks from time series data with probabilistic model checking. pages 26–32. cited By 7.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5.

Filiol, E. i irolla, p.(2015).(in) security of mobile banking... and of other mobile apps*.

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2015a). Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security Conference*, pages 513–527. Springer.

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. (2015b). Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA. IEEE Press.

Li, L., Bartel, A., Klein, J., and Le Traon, Y. (2014). Automatically exploiting potential component leaks in android applications. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 388–397. IEEE.

Liu, F., Cai, H., Wang, G., Yao, D. D., Elish, K. O., and Ryder, B. G. (2017). Mr-droid: A scalable and prioritized analysis of inter-app communication risks. *Proc. of MoST*.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016). Download malware? no, thanks: How formal methods can block update attacks. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, FormaliSE '16, pages 22–28, New York, NY, USA. ACM.

Milner, R. (1989). *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

Santone, A. (2011). Clone detection through process algebras and java bytecode. pages 73–74. cited By 10.

Spreitzenbarth, M., Echtler, F., Schreck, T., Freling, F. C., and Hoffmann, J. (2013). Mobilesandbox: Looking deeper into android applications. In *28th International ACM Symposium on Applied Computing (SAC)*.

Stirling, C. (1989). An introduction to modal and temporal logics for ccs. In *Concurrency: Theory, Language, And Architecture*, pages 2–20.

Wei, F., Roy, S., Ou, X., et al. (2014). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM.

Xu, K., Li, Y., and Deng, R. H. (2016). Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264.