Assisting Developers in Preventing Permissions Related Security Issues in Android Applications

Mohammed El Amin Tebib[1(✉)], Pascal André[2], Oum-El-Kheir Aktouf[1], and Mariem Graa[3]

[1] Univ. Grenoble Alpes, Grenoble INP, LCIS, Valence, France
{mohammed-el-amin.tebib,oum-el-kheir.aktouf}@lcis.grenoble-inp.fr
[2] LS2N CNRS UMR 6004 - University of Nantes, Nantes, France
pascal.andre@ls2n.fr
[3] IMT ATLANTIQUE, Nantes, France
mariem.graa@telecom-bretagne.eu

Abstract. Permissions related attacks are a widespread security issue in Android environment. Permissions misuse enables attackers to steal the application rights and perform malicious actions. While most of the existing solutions are advocated from end-users perspective, we take in this paper the developers perspective because security should be a software design concern. We propose a formal specification covering the permissions use by the current developers of Android applications, who are almost a third party developers. We underline a set of security properties. Then, we formally verify them by applying a Model Driven Reverse Engineering approach that enables abstraction and property verification. We implement the analysis approach as an IDE plug-in called PermDroid. Finally, we show the applicability of our approach through a case study.

Keywords: Android · Development · Security · Privacy · Permissions · IDE · MDRE

1 Introduction

Smartphones with Android platform are the most used ones with a market share of 72.6% (May 2020)[1]. According to CVE details[2], the official MITRE datasource for Android vulnerabilities, 2020 and the beginning of 2021 have witnessed the most significant increase of Android security threats, which could lead to several security attacks such as: collusion, privilege escalation, and ransomware attacks. While most of existing solutions are advocated from the end-user perspective,

[1] https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201905-202005.

[2] https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224.

R. Adler et al. (Eds.): EDCC 2021 Workshops, CCIS 1462, pp. 132–143, 2021.
https://doi.org/10.1007/978-3-030-86507-8_13

like anti-malwares [3], system configurations [12,20] and user-friendly tools [23], these security flaws are almost introduced at the development stage [13]. Developers often do not prioritize security during mobile applications development for various intrinsic and extrinsic reasons such as: (1) *lack of security skills*; (2) *scope of work*: developers leave this issue to security specialists, (3) *hardness*: security is a wide and difficult domain that slows down delivering the product in time. Despite real investment, security policies are therefore manually encoded into the application, which is a dangerous practice that may cause security breaches [2].

Among the security issues at development stage, we focus in this paper on permission related ones. Indeed, permissions are a main security concept to ensure the privacy protection, and restrict the access of third party libraries to the system resources. A recent experimentation study [19] realized on 574 GitHub repositories of open-source Android apps, showed that permissions-related issues are still a frequent phenomenon in Android apps. Android provides an official documentation to explain how to properly use permissions[3]. However, due to the continuous changes of permissions number and specification, this documentation becomes hardly readable for developers, leading to different security issues and drawbacks [11] such as: (i) Wrong permission usage: due to some permissions similarities, developers may intentionally use wrong permissions, *e.g.*: the use of *ACCESS_COARSE_LOCATION* instead of *ACCESS_FINE_LOCATION* (ii) Permissions over-privilege: a widespread phenomenon that occurs when the application declares more permissions than those actually used. The unused permissions can be exploited by hackers to perform malicious actions, especially ransomwares, (iii) Permissions under-privilege: occur when the application requires more permissions than those declared, which does not conform with the transparency principle that each developer should respect, (iv) Unprotected API: occurs when developers forget to add an exception handler to some API methods, which may throw exceptions.

In this paper we investigate the respect of permission security guidelines by the current developers of Android applications. We develop a formal framework for analyzing permission security issues like permissions naming conflicts, unauthorized access during component invocation, privilege escalation and permission over-privilege. The main contributions of this paper are: 1) A model based formal verification approach to assist third party developers in preventing the undesired permissions related behaviours that compromise the privacy of the application. 2) An implementation of the proposed approach in the form of an extensible IDE based tool (Eclipse, intellIJ, AndroidStudion) called **PermDroid** that identifies and prevents permission security risks and vulnerabilities related to the Android permission system. This tool analyzes up to the last Android API version 30. The rest of the paper is organized as follows. Section 2 presents the major permission security evolutions in Android versions. Section 3 describes the security mechanisms of Android with a special focus on permissions. We specify a set of security properties and guidelines in Sect. 4. The Analysis of these properties through an MDRE approach is presented in Sect. 5 and the experiments on a

[3] http://developer.android.com/sdk/api_diff/30/changes.

case study are provided in Sect. 6. Section 7 considers related works and finally, Sect. 8 concludes with a summary of our contributions and provides tracks for future work.

2 Android Permission System Evolution

The reader will find an interesting introduction to the Android Permission Model in [4]. In this section we quickly overview the most significant evolutions related to the Android permission system[4]. Table 1 summarizes the permissions-related security features categorized per Android version.

Table 1. Evolution of Android permission system

Ver. num	Version name	Security features
4	Kit kat	Permission groups
6	Marshmallow	Runtime permissions
8	Oreo	New access control mechanism
10	Ice cream sandwich	Permissions updates
11	Android 11	Permissions auto revoke

Since the creation of the first version of Android on 2008, its permission system has been under many studies related to Android security and privacy. The concept of *Permission* is introduced in the 4th version of Android "Kit Kat". Each application cannot access a sensitive information (*e.g.* contacts, SMS, location...) without having the required permissions. Those permissions could only be granted by the user at the installation time. Later in version 4, permissions groups were introduced to add a logical grouping of permissions sharing the same characteristics. If one of the permissions belonging to the same group is granted, the other permissions are automatically granted. The most significant update occurred on 2016. The user could avoid granting all permissions at install time. Indeed, the version (v6, API 23) brought large changes to the permission model enabling to partially check and grant permissions. Permission of normal category could be granted at install time and those of dangerous category could be checked and activated at run-time only; permission can be revoked later [1]. Last but not least, in the version 10, Android provides a modular permission controller that enables to update privacy policies and UI elements (*e.g.* policies and UI related to granting and managing permissions). Android 11 recently added the support of auto revoke apps, where the new Permissions Controller module can automatically revoke runtime permissions for Apps that haven't been used for an extended period of time. Apps targeting SDK 30 or higher have

[4] https://www.techrepublic.com/article/ios-and-android-security-a-timeline-of-the-highlights-and-the-lowlights/.

auto revoke enabled by default, while apps targeting SDK 29 or lower have auto revoke disabled by default[5].

Many evolutions made Android a more secured environment. Without understanding these evolutions, it is getting harder for developers to design secured Android application with respect to the permission security guidelines. A deep and continuous analysis, documentation and understanding are required by different stakeholders, especially developers.

3 Permission Security Model

Using the expressive Z notation [21], we formalize the permissions concepts in Android application. Formalization enables to understand the essence of the permission system. Besides, it enables to state security properties we ought to verify in Sect. 4 and check on Android apps in Sect. 5. Due to space limitation, a detailed specification is provided in a web appendix[6].

We consider Android applications made of *Components* that communicate via *Intents*. Basically, components are categorized into two families: 1) foreground components such as activities and 2) background components such as Services, Broadcast Receivers, and Content Providers. Applications are made of components ($cApp$), which can be part of several applications, and intents ($iApp$). Components can be source or target of intents ($csrcInt$, $ctargInt$).

In Android applications (defined by the Z schema AndroidApps (See Footnote 6), *Permissions* are declared in the manifest.xml configuration file, and required at different stages: (1) system APIs interactions, (2) database access, (3) message passing system via intents, (4) invocation of specific protected methods in public APIs and (5) content provider data access. They also have different protection levels. The Z basic types assume the existence of an abstract set of permissions and the free type provides the permission categories.

$$[PERMISSION]$$
$$Category ::= normal \mid dang \mid sig \mid sos$$

Permissions are declared per application ($permApp$), components ($permComp$) and intents ($permIntent$). Component permissions are declared by components ($cpermDec$) and required by active components ($cpermReq$). Passive components provide read and write access permissions ($rpermDec$, $wpermDec$).

[5] https://source.android.com/devices/architecture/modular-system/permissioncontroller.

[6] https://pagesperso.ls2n.fr/~andre-p/download/androidPerm.pdf.

AndroidPerm

$$
\begin{array}{l}
AndroidApps \\
Permissions : \mathbb{P}\, PERMISSION \\
decPermLevel : PERMISSION \nrightarrow Category \\
permLevel : PERMISSION \rightarrow Category \\
permApp : Applications \leftrightarrow Permissions \\
permComp : Components \leftrightarrow Permissions \\
permIntent : Intents \leftrightarrow Permissions \\
cpermDec : Components \leftrightarrow Permissions \\
cpermReq : CompAct \leftrightarrow Permissions \\
rpermDec, wpermDec : CompPas \leftrightarrow Permissions \\
\hline
\mathrm{dom}\, decPermLevel \subseteq Permissions \land \\
permLevel = (Permission \rightarrow \{normal\}) \oplus decPermLevel \\
\mathrm{ran}\, permApp \cup \mathrm{ran}\, permComp = Permissions \land \mathrm{ran}\, permIntent \subseteq Permissions \\
permComp = cpermDec \cup cpermReq \\
(rpermDec \cup wpermDec) = (CompPas \lhd cpermDec)
\end{array}
$$

Each permission belongs to one *Category* by *permLevel* ($\rightarrow$ is a total function). This category can be explicitly defined by *decPermLevel* (it is optional since $\nrightarrow$ is a partial function) and only the considered permissions have one $\mathrm{dom}\, permLevel \subseteq Permissions$. If no category is explicitly given, the level is *normal* $((Permission \rightarrow \{normal\}) \oplus decPermLevel)$. All considered *Permissions* are associated to applications or components by $(permApp \cup \mathrm{ran}\, permComp = Permissions)$. Intents permissions are related to them by $permIntent \subseteq Permissions$. Component permissions are declared or required $(permComp = cpermDec \cup cpermReq)$ but only active components require permissions. Passive components provide read and write access permissions $(rpermDec, wpermDec)$ of the declared permissions of passive components $((rpermDec \cup wpermDec) = (CompPas \lhd cpermDec))$. In the next section we formally specify security properties using the above formal descriptions.

4 PermDroid Security Properties

Referring to standards such as CIA (Confidentiality, Integrity, Availability) or AAA (Authorization, Authentication, Accounting) [9], we target confidentiality and authorization properties. Again for sake of space, only two properties are used in this paper but others are given in the web appendix (See Footnote 6). We refer here to [16] that investigated the mistakes committed by developers.

Component Invocation. As a component could perform sensitive actions, it should be protected against unauthorized accesses. *(p1) Two interacting components must have compatible permissions. Every required permission of a called component must be fulfilled by the caller component.*

P1

$$\begin{array}{|l} \textit{AndroidPerm} \\ ca, cp : \textit{COMPONENT} \\ \hline (ca, cp) \in \textit{interactions} \Rightarrow \textit{cpermReq}(\!| \{co\} |\!) \subseteq \textit{cpermDec}(\!| \{ca\} |\!) \end{array}$$

Unprotected Components - Privilege Escalation. Exported components can be accessed by other applications including malicious apps. Referring again [16], a significant number of non protected exported components was found in on-line Android open source projects (19039 components overall 8749 application).

(P2) An exported component must declare permissions to be protected.

P2

$$\begin{array}{|l} \textit{AndroidPerm} \\ \hline \textit{cpermDec}(\!| \textit{expComp} |\!) \neq \emptyset \end{array}$$

Many related research studies [6,7,12] explore the dynamic behaviour of interacting applications to find permission related security flaws. Based on static analysis, we could only (1) determine the requested permissions declared in the manifest configuration file, (2) generate permissions used by the application through inspecting the permission related APIs, (3) inspect methods involving sending and receiving intents, (4) and methods involving the management of content providers. Whereas dynamic analysis could assist in (1) handling the dynamic loading of classes from embedded jar or apk files, (2) handling Java reflection (used by more than 57% of Android apps [15]).

Z/EVES System [17] enables not only syntax and type checking verification, but also the proof of operation assertions (pre/post conditions vs the state schema invariants) and theorems. The effective verifications are not provided here but the formalisation is an input for the implementation, where the rules will be written in OCL which is inspired from Z.

5 PermDroid Design and Implementation

In Sect. 4 we formalized five specific security related permissions issues coming from: permissions naming violations, unprotected components, components invocation, over-privileged permission use, and unprotected implicit intents. To assist

developers (especially third party ones) to automatically prevent these issues in their implementations, we propose in this section a model-based approach using a security meta-model inspired from the formal model presented in Sect. 3 with some additional technical aspects[7]. The proposed approach is given in Fig. 1. It is based on three main steps (see Fig. 1): (1) Reverse engineering, (2) Model to Model (M2M) transformations, and (3) Analysis phase.

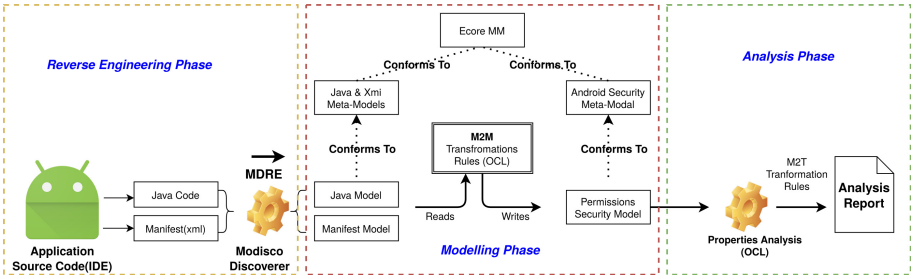

Fig. 1. PermDroid architecture

Reverse Engineering Phase (Model Discovery). Reverse engineering provides abstract models that capture the pertinent information for reasoning purposes. We used MoDisco[8] tool to perform the reverse engineering process. Basically, MoDisco provides a graphical representation of the program Abstract Syntax Tree AST[9] which makes the corresponding models it generates correct and perfectly conform to the source code. In addition, MoDisco supports this process for Java and XML based applications, which is relevant to study Android applications. We developed a set of Java scripts that take as input the manifest and the Java source files of an Android application, and generate as output the xmi corresponding model.

Model Transformation Phase. At the end of the first step, we apply a M2M transformation process using ATL[10] tool which is based on the OCL formal language. As a result of the model transformation process, we obtain the Android security model used for the analysis phase. A schematic view showing the whole model-driven chain is presented in phase 2 of Fig. 1.

The transformation engine we implemented is composed from a set of rules. Each one serves to *read* a specific part from the source model "Java or manifest models" and *write* the corresponding element in the target model following a well defined semantic. The following small OCL code shows the main rule of the transformation script that aims to generate an instance of *application* element

[7] See the web appendix (See Footnote 6).
[8] https://wiki.eclipse.org/MoDisco.
[9] https://www.vogella.com/tutorials/EclipseJDT/article.html.
[10] https://www.eclipse.org/atl/.

structure presented in the meta-model of Fig. 1 from the *root* XML element of the manifest file.

```
--@nsURI MM=www.eclipse.org/MoDisco/Xml/0.1.incubation/XML
--@path MM1=/safetyProperties/metamodels/androidapp.ecore
rule mainRule {
  from   manifest : MM!Root (manifest.name="manifest")
  to     securityModel : MM1!Application (
             name <- manifest.name,
             permissions <- MM1!Permission.allInstances(),
             components  <- MM1!Component.allInstances())}
```

Security Properties Analysis Phase. For the analysis phase, we implemented using ATL a set of rules called *helpers* (like methods in Java but with a formal constraint format) to implement the security properties we specified in Sect. 4. To make clear the idea of how to use OCL for the analysis phase, we present in the following a simple part of the implemented helpers, but we do not put the whole rules due to space limitation.

```
helper def: permission_name_conflict(): Boolean =
  if ((thisModule.normalPermissions -> intersection(thisModule.
       dangerousPermissions))-> isEmpty())
  then true       else false       endif;
```

The permission_name_conflict() is an example of helper that is used to implement permission property (P1). It parses the instance security model representing the Android application under analysis, to inspect if there is such permission with normal protection level that is also declared for a second time as dangerous permission. If the helper return type is true, our tool will notify the developer. We give an example on the use of PermDroid at the end of Sect. 6.

6 Experimentation

To experiment the ongoing work on PermDroid tool, we select a simple open source application called ***Telegram*** extracted from F_Droid[11], a famous repository for Android open source applications. It represents a messaging app with a main focus on speed and security. We add some modifications to the application in order to inject the security flaws we want to raise at the analysis phase. The goal is to validate the followed steps based on a simple case study.

Reverse Engineering (RE) Phase. As mentioned before, in the RE phase PermDroid will have as input the Android application code files (manifest.xml and Java classes). MoDisco provides an abstract representation of the manifest file. It exposes the *root* instance that represents the composite element by which we can access children items such us: components, intents, permissions and their attributes. For sake of space, we add on a the web appendix a detailed screenshot

[11] https://f-droid.org/app/org.telegram.messenger.

that presents the generated model representing the manifest configuration file of ***Telegram*** application.

Modelling Phase. Once the corresponding ***Telegram*** demo app models are generated, PermDroid launches the transformation process to generate the application security model displayed in Fig. 7 of web appendix, where types specified on our security meta-model are presented on the left side, and the existing instances for each type are presented on the right side. Based on these representation, we notice that the demo app is composed only from background components: 15 services, 18 broadcast receivers and 3 content providers.

Analysis Results. The final step is to launch the analysis of the security properties. Figure 2 shows the analysis results raised by PermDroid tool. The analysis report raises the unsatisfaction of the security property related to naming conflicts. It indicates that *perm1* = *"BIND_TELECOM_CONNECTION_SERVICE"* with *Nrml_perm* is duplicated in: *"Content Provider":"notification_image_provider"* with "Dangerous" protection level.

Fig. 2. Analysis results of *Telegram* demo app

7 Related Works

During the last decade, many research works exploring security gaps in Android permission system have been published. These works can be classified into two main classes.

System Analysis and reconfiguration approaches that mainly consider the weaknesses that reside on the system on one side, and the user knowledge limitations with regards to Android permission granting on the other side. The goal is to make the system environment safer by putting the malware applications apart from the system. In [14] a formal model of the Android permission framework using high level Petri nets is presented. It defines the complex relationships among different levels of permissions such as overall application-level permissions and component level permissions. In the same line, many other works are based on formal models and analysis [8,12] to verify the security mechanisms

of Android. These works generally differ on the analysed Android version: [12] before version 6 where the permission granting occurs always at install time, and [8] after version 6 where permissions could be granted dynamically at run-time.

Other solutions basically require modification of the Android framework or apps' implementation logic. As an example we can cite [20] and [18] which change the permission management mechanism, and propose a new Android security model and isolation strategy respectively. These modifications will impose to a developer to change the usual way of manipulating the manifest configuration file, which makes this solutions too hard to be adopted by developers. *As we can see, most of the research related solutions are proposed from end users perspective. We conduct our research from a different design methodology. We tackle developers perspective because detecting permission security issues in an early stage will evade unnecessary vulnerabilities, and prevent the potential for abuse by SDKs.*

Developer's perspective approaches aim to automate the assistance to developers who are almost not aware about security issues introduced during the development stage. There are few related works aiming to assist developers identifying permission related security issues. A big parts of these works focuses on over-privilege permissions issue; they propose integrated IDE solutions to help developers make better decision about the permission use. The goal is to help the developer in building apps with the least required permissions, like works in [5,10,22] that statically analyse the manifest and the Java code, and separate between two sets of permissions: (1) the declared permissions set and (2) the used permission set. The idea is that to reach least privilege principle, the application should not declare more permissions than those used. In the same line [24] gives a more accurate analysis and adds analysis capacities to cover underprivileged permissions and unprotected APIs in addition to over-privileged ones. The main limitations of these tools are that the used permission-function pairs are generated for only specific Android API levels (2.2 for [22] and 5 for [5]). Furthermore, all of them are static analysis tools, while the big part of used permissions will be granted during run-time. This limitation will reduce the accuracy of *used permission* set content.

Our approach belongs to this class of solutions. We benefit from formal methods to perform high level analysis for developers. We follow a similar methodology as done by [24], *where they benefit from system analysis studies like* [3] *to provide to developers an automatic solution helping them detecting if their code does not respect the least privilege principle.*

8 Conclusion and Future Work

This work constitutes a basis for the development of a well-defined formal framework called PermDroid that assists developers in preventing permission security related-issues. The starting point was to define a formal specification that covers the minimum number of concepts related to permissions used in the context of a

single application. This formalization helped us to define some security properties that are verified using an automated approach based on MDRE. We finally experimented the implemented architecture through a simple case study.

Ongoing and future works consider the extension of studied security properties, and the integration of dynamic analysis to our approach. This will contribute to handle dynamic loading of classes and Java reflection. The expected final outcome of our work is a flexible integrated tool that implements our approach and can be integrated in several IDEs. Finally, we aim to experiment our approach on a large data set, and evaluate its usability through organizing controlled practical dev sessions.

References

1. Almomani, I.M., Khayer, A.A.: A comprehensive analysis of the Android permissions system. IEEE Access **8**, 216671–216688 (2020). https://doi.org/10.1109/ACCESS.2020.3041432
2. Armando, A., Carbone, R., Costa, G., Merlo, A.: Android permissions unleashed. In: 2015 IEEE 28th Computer Security Foundations Symposium, pp. 320–333. IEEE (2015)
3. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 217–228 (2012)
4. Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the Android permission system. Formal Aspects Comput. **30**(5), 525–544 (2017). https://doi.org/10.1007/s00165-017-0445-z
5. Bello-Ogunu, E., Shehab, M.: PERMITME: integrating Android permissioning support in the IDE. In: Proceedings of the 2014 Workshop on Eclipse Technology eXchange, pp. 15–20 (2014)
6. Betarte, G., Campo, J., Cristiá, M., Gorostiaga, F., Luna, C., Sanz, C.: Towards formal model-based analysis and testing of Android's security mechanisms. In: 2017 XLIII Latin American Computer Conference (CLEI), pp. 1–10. IEEE (2017)
7. Betarte, G., Campo, J., Luna, C., Romano, A.: Formal analysis of Android's permission-based security model. Sci. Ann. Comput. Sci. **26**(1), 27–68 (2016)
8. Betarte, G., Campo, J., Luna, C., Sanz, C., Gorostiaga, F., Cristiá, M.: A formal approach for the verification of the permission-based security model of Android. CLEI Electron. J. **21**(2) (2018)
9. Buchanan, W.: Introduction to Security and Network Forensics. Taylor & Francis (2011). https://books.google.fr/books?id=8uzM63AYi_MC
10. Chester, P., Jones, C., Mkaouer, M.W., Krutz, D.E.: M-Perm: a lightweight detector for Android permission gaps. In: 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 217–218. IEEE (2017)
11. Fang, Z., Han, W., Li, Y.: Permission based android security: issues and countermeasures. Comput. Secur. **43**, 205–218 (2014)
12. Fragkaki, Elli, Bauer, Lujo, Jia, Limin, Swasey, David: Modeling and enhancing Android's permission system. In: Foresti, Sara, Yung, Moti, Martinelli, Fabio (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 1–18. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33167-1_1
13. Guo, W.: Management system for secure mobile application development. In: Proceedings of the ACM Turing Celebration Conference, China, pp. 1–4 (2019)

14. He, X.: Modeling and analyzing the Android permission framework using high level Petri Nets. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 232–239. IEEE (2017)
15. Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M.: Slicing droids: program slicing for smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1844–1851 (2013)
16. Jha, A.K., Lee, S., Lee, W.J.: Developer mistakes in writing Android manifests: an empirical study of configuration errors. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 25–36. IEEE (2017)
17. Saaltink, M.: The Z/EVES system. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–85. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0027284
18. Sadeghi, A., Jabbarvand, R., Ghorbani, N., Bagheri, H., Malek, S.: A temporal permission analysis and enforcement framework for Android. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, pp. 846–857. ACM, New York (2018)
19. Scoccia, G.L., Peruma, A., Pujols, V., Malavolta, I., Krutz, D.E.: Permission issues in open-source Android apps: an exploratory study. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 238–249. IEEE (2019)
20. Seo, J., Kim, D., Cho, D., Shin, I., Kim, T.: FLEXDROID: enforcing in-app privilege separation in Android. In: NDSS (2016)
21. Spivey, J.M.: Z Notation - A Reference Manual, 2nd edn. Prentice Hall International Series in Computer Science. Prentice Hall (1992)
22. Vidas, T., Christin, N., Cranor, L.: Curbing Android permission creep. In: Proceedings of the Web, vol. 2, pp. 91–96 (2011)
23. Wu, S., Liu, J.: Overprivileged permission detection for Android applications. In: ICC 2019–2019 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2019)
24. Xu, G., Xu, S., Gao, C., Wang, B., Xu, G.: PerHelper: helping developers make better decisions on permission uses in Android apps. Appl. Sci. **9**(18), 3699 (2019)